VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO PODPORU ANALÝZY RIZIK V INFORMAČNÍ BEZPEČNOSTI

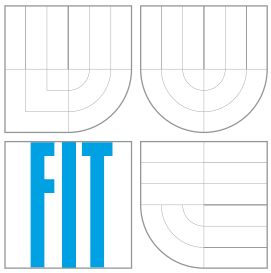
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN PLÍŠEK

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO PODPORU ANALÝZY RIZIK V INFORMAČNÍ BEZPEČNOSTI

TOOL FOR RISK ANALYSIS SUPPORT IN INFORMATION SECURITY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN PLÍŠEK

VEDOUCÍ PRÁCE
SUPERVISOR

doc. RNDr. JITKA KRESLÍKOVÁ, CSc.

BRNO 2012

Abstrakt

Tato diplomová práce se zabývá vytvořením nástroje pro analýzu rizik v informační bezpečnosti. Nejprve je nastudován proces vývoje bezpečného softwaru SDL a poté také teorie o analýze rizik pomocí chybových stromů. Na základě těchto znalostí je systém navržen a poté i implementován. Dále je předvedena práce s aplikací na typovém příkladu a popsáno možné využití nástroje v praxi. Na závěr jsou uvedeny možnosti budoucího rozšíření aplikace.

Abstract

The master thesis deals with the development of the tool for risk analysis support in information security. At first we perform a theoretical basis for security development of lifecycle process (SDL). Afterwards the theory of risk analysis based on fault tree analysis is described. Considering this knowledge base system was designed and implemented. Next chapter describes the best practice refer to the typical example of use and presents the potencial using of this tool in practice. Final chapter deals with the possibility of future expansion of this application.

Klíčová slova

Životní cyklus vývoje bezpečného softwaru, SDL, Analýza rizik, Hodnocení rizik, Data Flow Diagram, DFD, Model STRIDE, Analýza pomocí chybových stromů, FTA

Keywords

Security Development Lifecycle, SDL, Risk Analysis, Risk Assessment, Data Flow Diagram, DFD, STRIDE model, Fault Tree Analysis, FTA

Citace

Martin Plíšek: Nástroj pro podporu analýzy rizik
v informační bezpečnosti, diplomová práce, Brno, FIT VUT v Brně, 2012

Nástroj pro podporu analýzy rizik v informační bezpečnosti

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. RNDr. Jitky Kreslíkové, CSc.

.....
Martin Plíšek
21. května 2012

Poděkování

Rád bych na tomto místě uvedl poděkování doc. RNDr. Jitce Kreslíkové, CSc. za odborné vedení při vypracování této práce.

Obsah

Kapitola 1

Úvod

V dnešní době je bezpečnost něco, co nemůže být podceňováno. Svět je propojený pomocí internetu a v podstatě každý, kdo má zařízení připojené na tuto síť, může komunikovat s kýmkoliv. Toho však mohou zneužít různé kriminální frakce a mohou tak získat nepřeberné množství informací o uživatelích, jako například osobní informace, telefonní čísla, e-mailové adresy, ale třeba i čísla kreditních karet či přihlašovací údaje k internetovému bankovnictví.

Proti takovým útokům se snaží vývojáři a návrháři softwaru bojovat lepším přístupem k vývoji softwaru. Jedním z vývojových přístupů, který je trendem při vývoji softwaru ve společnosti Microsoft, je proces Security Development Lifecycle (SDL). První část tohoto textu se zabývá jeho popisem. Je popsán celý cyklus od počátečních fází návrhu až po vydání produktu a jeho údržbu.

Druhá část práce je zaměřena na analýzu rizik. Proces SDL obsahuje také část zabývající se analýzou a tato část byla v textu ještě rozšířena o pravděpodobnostní analýzu. Ta je založena na speciálních modelovacích strukturách, tzv. chybových stromech.

Z těchto dvou teoretických základů je poté vytvořen návrh na propojení těchto dvou oblastí znalostí. Návrhová kapitola poté pokračuje návrhem konkrétního softwarového nástroje, který by sloužil pro podporu analýzy rizik v informační bezpečnosti. Z návrhu bylo nutné vytyčit, co bude pro uživatele přednější pro prezentování a lepší orientaci v aplikaci. Z této myšlenky poté vycházel návrh grafického uživatelského prostředí, vnitřních datových struktur a chování samotné aplikace.

Obsáhlým celkem založeném na návrhu je implementační kapitola. Ta popisuje postup při vývoji celého systému a popisuje i problémy, se kterými jsem se musel potýkat, případně mechanismy, díky kterým je aplikace rychlejší a stabilnější.

V závěrečné kapitole je prezentována práce s jednoduchým modelem a je demonstrován postup při analýze pomocí implementovaného nástroje. Kapitola dále obsahuje informace o možném využití aplikace v praxi a o možnostech dalšího vývoje a rozšiřování funkčnosti.

Tento text volně navazuje na semestrální projekt. V semestrálním projektu byla uvedena celá teoretická část zabývající se popisem procesu pro vývoj bezpečného softwaru (2), kapitola zabývající se analýzou rizik (3) a semestrální projekt byl uzavřen kapitolou popisující spojení všech teoretických znalostí do jednoho celku (4.1). Text od této kapitoly dále již zcela zasahuje do diplomového projektu.

Kapitola 2

Životní cyklus vývoje bezpečného softwaru

Tato kapitola se zabývá tvorbou softwarových produktů pomocí procesu, který byl vyvinut společností Microsoft během několika posledních let a hlavně na základě zkušeností s bezpečnostními incidenty ve starších verzích OS Windows, SQL serveru a IIS web serveru. Interní název procesu je Security Development Lifecycle (dále jen SDL) a jak již název napovídá, tak je zaměřen především na stránku bezpečnosti. Manažeři v Microsoftu si položili otázku: „Proč je nutné se zabývat bezpečností?“ Odpověď je velice jednoduchá. Dříve byly počítačové systémy používány jako samostatné stanice a málokdy byly připojeny do sítě internet. Avšak s postupem času a rozvoje a rozmachu internetu bylo čím dál více počítačů propojených do sítí. V současnosti můžeme říci, že svět je pomocí internetu propojený více než kdy dříve a očekává se, že bude propojený ještě mnohem více.

Další věcí, která se z hlediska napadnutelnosti informací změnila, je zaměření útoků více na samotné aplikace nebo databáze. Dříve byly prováděny spíše útoky na samotné počítače jako takové, konkrétně napadání samotného operačního systému počítače. V dnešní době jsou však operační systémy mnohem více zabezpečené, a proto se útoky více zaměřují na aplikace samotné. Z toho tedy vyplývá, že je nesmírně důležité již při vývoji softwaru dbát na možné vlastnosti aplikace, kterých by mohl útočník využít ve svůj prospěch.

Zavedení procesu SDL nemůže mít pouze výhody z hlediska implementace, ale je nutno tuto myšlenku vhodně podat i pro obchodní zájmy softwarové firmy. Jelikož se SDL zabývá z velké části bezpečností, tato myšlenka má v dnešním „nebezpečném“ světě velký obchodní potenciál. Více bezpečný software se tedy bude lépe prodávat.

Tato kapitola vychází z knihy popisující SDL [5].

2.1 Důležité pojmy pro SDL

V této části jsem se zaměřil na vymezení několika pojmů, které budou na následujících stranách textu často zmiňované. Jsou to pojmy soukromí a bezpečnost, spolehlivost a kvalita softwaru.

2.1.1 Soukromí a bezpečnost

Mnoho lidí nahlíží na soukromí (privacy) a bezpečnost (security) jako na jeden a ten samý problém. Soukromí však chápeme hlavně v souladu s jistou bezpečnostní politikou, kdežto bezpečnost je chápána jako způsob vynucování takové politiky.

Pro přehlednost si uveďme příklad. Představme si, že jsme v restauraci, kde jsou oddělené toalety na pánské a dámské. Na každých dveřích je silueta (panenka/panáček) indikující, která toaleta je určena pro které pohlaví. Silueta tedy indikuje politiku, kdo do dané místnosti může a nemůže vstoupit. Na dámských toaletách mají soukromí ženy, na pánských zase muži. Tato politika ale nezabezpečuje před vstupem osoby opačného pohlaví. Pokud však přidáme na každé dveře toalet zámek, zvýšíme bezpečnost a pomůžeme tím vynutit politiku pro soukromí.

2.1.2 Spolehlivost

Spolehlivost je dalším aspektem, který má vliv na bezpečnost. Většinou je spolehlivost podložena dokumentem, kdy se společnost zavazuje k určité míře spolehlivosti, případně rychlosti odezvy na vzniklý problém. Tím dokumentem je tzv. Service Level Agreement, někdy též označovaný zkratkou SLA. Ke zvyšování spolehlivosti může být v softwaru implementováno několik funkcí.

- Zápisy chybových auditů (sledování pádů a bezpečnostních incidentů). Zde je nutno brát v potaz potencionální riziko, kdy bude log plný a nebudou se logovat další bezpečnostní incidenty, například vynutit zastavení aplikace při naplněném logu.
- Ošetřit případné pády služeb. Je nutno zajistit, aby služba, která selhala, byla znovu restartována a nebyla tak narušena funkčnost. Na druhou stranu je však toto také potencionální riziko, jelikož tento mechanismus dává útočníkovi šanci využít této slabiny. Proto se v praxi většinou zavádí buď povolení pouze tří restartů a poté již služba nenaběhne (3. pád je nahlášen administrátorovi) nebo se přidává a postupně zvyšuje prodleva mezi jednotlivými restarty.

2.1.3 Kvalita softwaru

Tento pojem v podstatě slučuje předchozí tři pojmy, jelikož kvalita je založena na vhodném vyvážení bezpečnosti, soukromí a spolehlivosti. Vztah kvality, bezpečnosti, soukromí a spolehlivosti je popsán obrázkem 2.1.

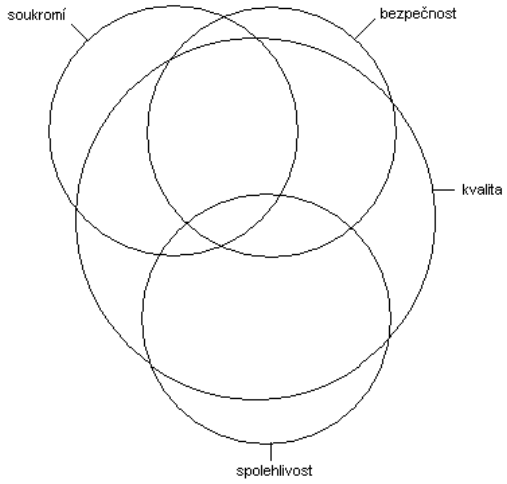
Zde jsou některé možnosti, jak se mohou dvojice výše uvedených vlastností střetávat v dobrém i špatném slova smyslu:

- Bezpečnost a soukromí. Implementací šifrování (bezpečnost) citlivých informací (soukromí).
- Bezpečnost a spolehlivost. Například DoS[1] útok je problémem jak bezpečnosti, tak spolehlivosti.
- Spolehlivost a soukromí. Při pádu aplikace (spolehlivost) může být v chybové hlášce obsažena citlivá informace (soukromí).

2.2 Metodiky tvorby softwaru

V této sekci jsem se zaměřil na rozdělení metodik tvorby softwaru a popsal jsem jejich základní vlastnosti.

[1] Denial of Service – zamezení služby



Obrázek 2.1: Vztah mezi kvalitou, bezpečností, soukromím a spolehlivostí [5]

2.2.1 Metodika „Více očí více vidí“

Originální anglický název této metodiky je *Given enough eyeballs, all bugs are shallow.* Pokud si volně přeložíme tento název z anglického jazyka, tak dostaneme následující tvrzení: „Pokud máme dostatek očí, pak jsou všechny chyby snadno nalezitelné.“ Metodika využívá velkého počtu lidí podílejících se na celém procesu vývoje daného softwarového produktu. Idea metodiky je, že pokud máme velké množství vývojářů a zároveň velké množství testerů, tak jsou všechny chyby v softwaru rychle identifikovány a opraveny.

Toto je typický vývojový cyklus pro softwarové projekty s otevřeným zdrojovým kódem, jejímž produktem jsou tzv. Open-source aplikace nebo aplikace, založené na licencích pro volné šíření či dokonce modifikaci.

Metodika má však také negativní dopad na samotný produkt. Jelikož je spousta lidí zapojených do procesu vývoje, je také spousta lidí, kteří vědí jak daná aplikace funguje a jaké má slabiny či zranitelnosti. Potom již není těžké, aby některý vývojář využil tyto znalosti k útoku na daný systém.

2.2.2 Vlastní vývojové metodiky

I když je spousta osvědčených metod, jak vyvíjet software, ne každé firmě to může vyhovovat a proto si firmy zavádějí vlastní vývojové cykly. Využívají přitom základních metod, jako například vodopádový model, spirálový model nebo CM[2] model.

SDL je cyklus zaměřený hlavně na bezpečnost a soukromí. Naproti tomu některé firmy využívají ostatní metody, jako například CMMI (Capability Maturity Model Integration), TSP (Team Software Process) a PSP (Personal Software Process). Ty jsou zaměřeny naopak na zlepšování kvality softwaru a konzistenci vývojového cyklu. Tyto metodiky nemají oproti SDL žádné specifické požadavky na bezpečnost.

[2]Capability Maturity

2.2.3 Agilní vývojové metodiky

Agilní metodiky jsou procesy založené na iteracích. Pomocí agilních metodik lze redukovat rizika při vývoji. Důležitá je v tomto přístupu odezva od zákazníků, čímž lze zajišťovat úpravy včas a efektivně. Velkou výhodou je lepší rozvržení času a predikce plánování celého cyklu.

2.2.4 Common Criteria (CC)

Nejedná se o metodiku, ale spíše o mezinárodní standard pro počítačovou bezpečnost. Dovoluje uživatelům definovat bezpečnostní požadavky, vývojářům specifikuje bezpečnostní atributy a 3. stranám poskytuje výsledky k porovnání se skutečným stavem. Tento standard nedefinuje požadavky na kvalitu kódu nebo designu.

CC definuje soubory požadavků nazývaných Evaluation Assurance Levels (EALs). EAL může nabývat hodnot v rozmezí 1 až 7. Čím vyšší je číslo EAL, tím je více potřeba času, úsilí a nákladů. Vyšší číslo EAL ještě nutně neznamená větší bezpečnost. Specifikace návrhu neobsahuje žádné důležité bezpečnostní detaily. Ty jsou uvedeny pouze v kódu, aby se zabránilo úniku dat.

2.3 Jednotlivé fáze procesu SDL

V této kapitole si postupně projdeme všechny fáze procesu SDL. Jednotlivé fáze korespondují s vývojovým cyklem softwaru a bezprostředně na sebe navazují. Hlavní podmínka SDL totiž je, že žádná fáze nemůže být vynechána, jelikož by tím došlo k narušení celého procesu vývoje bezpečného softwaru.

2.3.1 Fáze 0: Vzdělávání a vědomosti

Fáze vzdělávání a vědomosti je založena na tom, že všichni účatníci SDL budou informováni o novinkách a trendech ve vývoji. Jedná se především o bezpečnostní rizika, která se mohou postupem času měnit nebo se objevují úplně nové hrozby. Tím je tedy dáno, že musí firma pořádat pravidelné training programy a cvičení o bezpečnosti. Nutností je, aby se tohoto školení účastnili zaměstnanci s vysokým bezpečnostním vzděláním. Důraz při těchto programech je kladen hlavně na schopnost použít tyto nově nabyté dovednosti při práci. Toto se většinou ošetřuje zavedením kratkých cvičení nebo příkladů na konci prezentace. Pokud má navíc firma velký počet zaměstnanců, je nutné zavést systém distribuce studijních materiálů. Většinou se využívá vnitřní firemní síť intranet nebo jsou studijní materiály vloženy na firemní web.

Cílem školení není udělat ze zaměstnanců bezpečnostní experty, ale zvýšit jejich povědomí o bezpečnostních problémech. Firma by měla zajistit, aby takové školení probíhalo 1x ročně se 100% účastí a aby to bylo skutečně dodržováno.

2.3.2 Fáze 1: Započetí projektu

Úplně na začátku každého projektu je potřeba určit, zda se na danou oblast vztahuje politika procesu SDL, do jaké míry je potřeba SDL použít a zda vůbec má zavedení SDL při vývoji nějaké přínosy. Pro názornost si uveďme pár příkladů:

- Aplikace používaná v bussiness sféře (emailový nebo databázový server).

- Aplikace zpracovávající a ukládající osobní data (finanční nebo lékařský sektor).
- Aplikace pravidelně přistupující k internetu (emailový nebo IM klient).
- Aplikace se pravidelně aktualizuje.

Pokud aplikace popisuje alespoň jednu z výše popsaných, potom je pravděpodobně vhodné zvážit při vývoji SDL.

Druhým důležitým bodem je zvolení bezpečnostního poradce. Ten má na starosti dohled celého SDL a jeho cílem je úspěšné dokončení fáze *Konečného shodnocení bezpečnosti* (více v kapitole 2.3.10). Zaujímá také pozici jakéhosi prostředníka při komunikaci mezi vývojovým a bezpečnostním týmem. S vývojovým týmem navíc konzultuje modely návrhu a rizik. Jeho speciálním samostatným úkolem je detekovat potencionální chyby, které by mohly narušit bezpečnost nebo soukromí.

Dále je třeba vytvořit vedoucí tým bezpečnosti. Ve větších firmách to bývá skupinka lidí, ale v menší firmách může tuto funkci zastávat pouze jeden člověk. Funkcí týmu je komunikovat s vývojovým týmem a hlídat počet bezpečnostních chyb a chyb týkajících se soukromí. Tým je více zaměřen na technickou stránku věci, narozdíl od bezpečnostního poradce.

V této fázi je nutné zajistit, že systém sledování chyb (bugtracker) bude zahrnovat speciální kategorie pro chyby týkající se bezpečnosti a soukromí. Úkolem je sledovat nejen příčiny chyb, ale také následky, které může daná chyba způsobit. Tyto kategorie je navíc potřeba rozlišovat různými úrovněmi závažnosti chyb. Dle závažnosti a časového rozvrhu lze totiž buď chybu opravit (kritická chyba), zmírnit její následky (méně závažná chyba) nebo ji nechat neopravenou (chyba nemá větší dopad na bezpečnost a soukromí).

2.3.3 Fáze 2: Definice a dodržování osvědčených postupů

Popis této fáze rozdělíme na dvě části. Nejdříve se zaměříme na programovací techniky a poté si řekneme něco o napadnutelnosti produktu, dále jen AS[3].

Při programování by se měl vývojový tým řídit základními mechanismy „dobrého programování“. Dané úkoly by se měly postavit na více než jedné podmínce a mělo by se minimalizovat sdílení dat a proměnných mezi funkcemi. Konečná implementace by měla být zaměřena na koncového uživatele (user friendly GUI) a mělo by se předejít složitému nastavování funkcí aplikace. Samozřejmě je kladen důraz na ošetřování všech vstupů aplikace, tedy kontrola na neočekávané hodnoty a datové typy. Důležité je již během návrhu zavádět bezpečnostní požadavky a návrhy na funkčnost, jelikož všechny pozdější úpravy jsou neefektivní a většinou i nákladné na zdroje.

Nyní se pozastavme nad problémem napadnutelnosti produktu – *Attack Surface* (dále jen AS). AS je kód, který je přístupný jak uživatelům, tak i potencionálním útočníkům. Aplikace, která má velký AS, musí být precizně napsaná a intenzivněji testovaná. Důležité je zamyslet se, zda je některá funkce aplikace s velkým AS tak důležitá. Méně důležité funkce s velkým AS by měly být v základním nastavení vypnuty, aby se zabránilo postižení většího počtu uživatelů v případě potencionální chyby. Dále je potřeba si položit otázku, kdo potřebuje přístup k dané funkcionalitě a odkud k ní bude přistupovat. Například při síťovém přístupu není dobré se spoléhat pouze na bezpečnostní bránu (firewall), ale je dobré přidat další bezpečnostní vrstvu pro případ, kdy je brána vypnutá. Může to znamenat například zavedení privilegií pomocí seznamu řízení přístupu – ACL (ve Windows UAC[4], v *nix

[3] attack surface – míra napadnutelnosti softwaru (pojem je hojně využívaný v literatuře)
[4] User Access Control

systémech ACL[5]). Komplexně lze aplikaci rozdělit na jádro a na uživatelskou část. Jádro potom poběží s root právy a uživatelská část s uživatelskými právy. Z hlediska síťového přístupu bude jistě rozumnější zvážit použití protokolu TCP místo UDP.

2.3.4 Fáze 3: Hodnocení rizik

Fáze hodnocení rizik definuje úroveň vynaložení nákladů tak, aby byly splněny požadavky procesu SDL. Hodnocení bezpečnostních rizik slouží pro zjištění zranitelnosti a náchylnosti k útoku na produkt.

Pro hodnocení bezpečnostních rizik máme několik hledisek:

1. Na jakém OS aplikace poběží?
2. Je vyžadováno administrátorské heslo?
3. Využívá aplikace ACL?
4. Modifikuje aplikace AD[6] schéma?

Toto byl výčet základních obecně zvažovaných hledisek. My si však klademe na zhodnocení ještě další otázky. Musí se provést důkladná analýza AS, analýza v případě použití mobilního kódu a analyzování kompatibility s předchozími verzemi.

Pokud budeme hodnotit rizika týkající se soukromí, tak použijeme následující tří-úrovňové hodnocení:

1. úroveň Aplikace ukládá a převádí PII[7]. Může se týkat například věku, kdy aplikace není vhodná pro osoby mladší 12-ti let. Aplikace monitoruje uživatele. Aplikace instaluje nový software a mění asociaci datových typů.

2. úroveň Aplikace ukládá a převádí anonymní data výrobci sotfwaru nebo společnostem třetí strany.

3. úroveň To co nespadá pod 1. nebo 2. úroveň.

2.3.5 Fáze 4: Analýza rizik

Tato fáze následuje ihned po fázi zhodnocení rizik, ale jelikož je tato práce na analýzu rizik zaměřená, tak se jí budu věnovat v samostatné kapitole, konkrétně v kapitole 3.

2.3.6 Fáze 5: Vytváření bezpečnostní dokumentace, nástrojů a případů použití pro zákazníky

Tato fáze je důležitá zejména z hlediska použitelnosti. Použitelnost a bezpečnost mohou totiž někdy kolidovat a proto je důležité vytváření bezpečnostní dokumentace a nástrojů. Zákazníky je navíc potřeba naučit, jak mají software zavést a jak jej mají správně a bezpečně používat.

Vytvoření dokumentace případů použití se skládá z několika důležitých částí.

[5] Access Control List
[6] Active Directory
[7] Personally identifiable information

Instalační dokumentace Obsahuje například informace, jaké porty je potřeba otevřít v bráně firewall. Dále je zde zmíněna zpětná kompatibilita s předchozími verzemi. V případě použití šifrování je zde i informace o získávání certifikátů a bezpečnostních klíčů.

Dokumentace použití Jedná se v podstatě o uživatelský manuál. Může zde být například zmíněna informace o změně nastavení při přechodu na novou verzi (změna databáze).

Nápověda Nápověda by měla být úkolově orientovaná, což znamená mít ke každému úkonu jeden případ použití (best practices).

Vývojová dokumentace Tato dokumentace je důležitá hlavně v případě, kdy je část nebo celá aplikace poskytována jako API tříd a objektů. Potom je třeba mít ke každé funkci a volání metody popis, příklad a případné bezpečnostní upřesnění.

Dalším krokem je vytvoření nástrojů. Pro jednoduché zavádění softwaru do firem jsou vytvořeny nástroje pro usnadnění nastavení. Uživatelé si pomocí nástroje jednoduše mohou nastavit všechny bezpečnostní parametry. Jedná se především o různé instalátory s předdefinovaným nastavením nebo instalační balíky použitelné pro instalaci přes GPO[8] v doméně.

2.3.7 Fáze 6: Politika bezpečného programování

Postupem času se vyvinuly programovací praktiky a využívá se velké množství pomocných prostředků, které zamezují vznik chyb již přímo při vývoji produktu. Tato podkapitola popisuje, jak může programátor již v etapě implementace předcházet chybám.

Prvním bodem bezpečného programování je bezesporu použití nejnovějšího překladače pro překlad zdrojového kódu. Novější překladače mají lepší výpisy varování, využívají pokročilejší optimalizační mechanismy a umožňují použití nejrůznějších přepínačů při překladu.

Další bod navazuje přímo na předchozí. Pomocí přepínačů u překladače můžeme zapnout ochrany zabudované v překladači. Pro názornost uvedeme 3:

1. **Bezpečnostní kontrola bufferu** – kontroluje přetečení bufferu (přepínač `/GS`)
2. **Bezpečné ošetřování vyjímek** – ověřování, zda nějaká aplikace nenarušuje volání vyjímek (přepínač `/SAFESEH`)
3. **Kompatibilita s mechanismem bezpečného spouštění** – ověření kompatibility s DEP[9] funkcí v MS Windows (přepínač `/NXCOMPAT`)

Nad napsaným zdrojovým kódem můžeme využít nástroje analyzující zdrojový kód. Jsou to všestranné pomůcky, které však nikdy nemohou nahradit člověka. Většinou tyto nástroje slouží pro odhalení nebezpečných funkcí. Některé mohou odhalit některé typické chyby zdrojového kódu, ale nemohou odhalit chybu návrhu. Výhodou je, že pokud se odhalí nějaká závažnější chyba, potom je vylepšen vzdělávací program nebo dokonce celý SDL proces.

Další věcí je zamezení používání zakázaných funkcí. V podstatě to souvisí s předchozím bodem, který mohl takové funkce odhalit. Některé takové funkce mohou vykazovat nebezpečné chování nebo jsou snadno napadnutelné (neznámá velikost čtení z bufferu) a proto byly zakázány. K zamezení můžeme využít hlavičkový soubor `banned.h`.

[8] Group Policy Object
[9] Data Execution Prevention

V systémech využívající ACL je třeba redukovat napadnutelné programové konstrukce a vzory. Odstranit bychom například měli běžně používanou inicializaci objektu pro ACL (`NULL DACL`), která vytvoří objekt s prázdným ACL a není tak chráněný.

Poslední radou je použití kontrolního seznamu bezpečného kódování (Secure Coding Checklistu). Ten obsahuje všechny minimální požadavky na veškerý kód, který je kontrolován.

2.3.8 Fáze 7: Politika testování bezpečnosti

Celá fáze testování se rozděluje na více přístupů k testům a také testům na různých aplikačních úrovních. Je nutné se také zpětně vrátit k návrhu a zkontrolovat korespondenci návrhu se skutečnou aplikací.

Fuzz testování

Koncept fuzz testování je velice jednoduchý. Dáváme aplikaci data na vstup a sledujeme reakce na výstupu. Testování je založeno na znalosti principů aplikace, jak aplikace ukládá data a také jaké používá datové struktury. Většinou se tímto způsobem testují parsery. Existuje několik typů fuzz testů pro parsery:

1. **Fuzz test formátu souboru**: Pomocí fuzz testování je vytvořen soubor, který přijde na vstup aplikace. Podle SDL musí být pro každý formát souboru, který aplikace používá, použito alespoň 100000 různých souborů. Je potřeba zmapovat opravdu všechny typy souborů, nejen ty „uživatelské“, ale třeba i šifrované nebo digitálně podepsané. Při vytváření speciálních souborů je nutné dbát třeba i na CRC[10]. S parsováním souvisí také vysoké paměťové nároky, proto je nutno sledovat i toto vytížení zdrojů.

2. **Fuzz test síťových protokolů**: Pomocí nějakého nástroje je na vstup aplikace (v tomto případě port) zasílán datový proud paketů. Aplikace proud zpracuje a účelem testu je zkontrolovat korektnost výstupu. Testují se většinou protokoly TCP a UDP, vzdálená volání procedur (RPC) nebo roury. Speciálním případem je testování zopakování paketů. Testování musí probíhat v obou směrech, tedy jak z klienta na server, tak ze serveru na klienta.

Penetrační testování

Obecně se jedná o proces navržený k nalezení slabin v informačních systémech. S tímto testováním je nutno začít již v brzkých fázích testování, aby se odhalené závažné chyby podařilo včas opravit. Tento proces vyžaduje zkušenosti s testováním a znalost konkrétního programovacího jazyka.

Verifikace za běhu

Jedná se o specifické testování založené na matematických modelech. Testuje se pomocí speciálních aplikací a hledány jsou specifické typy chyb. Toto testování je velmi náročné na výpočetní zdroje a probíhá za běhu programu.

[10] Cyclic redundancy check – kontrolní součet

Kontrola a případná aktualizace modelu hrozeb

Na základě modelu hrozeb jsou prováděny testy a pokud se během testování zjistí, že je potřeba model aktualizovat, tak dochází k jeho rozšíření. Je zde kladen důraz především na riskantnější části aplikace a ty musejí být také důkladněji testované.

Přehodnocení AS aplikace

V předchozích fázích procesu SDL jsme zjistili rozsah AS. Testování však může odhalit, že se AS změnil. Například při opravování chyb mohlo dojít buď ke zlepšení nebo také ke zhoršení rozsahu AS. Proto je nutné provést revizi a na základě toho provést nezbytná opatření.

2.3.9 Fáze 8: Security Push

Jelikož se výraz *Security Push* vyskytuje hojně v literatuře a je běžně používaný, nebudu ho překládat do češtiny. Dalo by se říci, že to je fáze zhodnocení bezpečnosti produktu. Cílem této fáze je nalézt bezpečnostní chyby, identifikovat jejich příčinu a proces zakončit jejich opravou. Fáze ještě nutně nezajišťuje bezpečný software, protože na bezpečnost je nutno myslet již v předchozích fázích (obecně platí, že je lepší, když chyby nevznikají, než když se musí potom hledat a opravovat).

Tato fáze začíná v momentě, kdy jsou veškeré programovací práce kompletní a všechny funkce jsou implementované. Dá se říci, že se jedná o verifikaci softwaru. Finální beta testování nastává až po této fázi. Pokud produkt používá část staršího kódu (kde ještě nebyla vyžadována taková bezpečnost – nebyl zaveden SDL), potom je Security Push více zaměřen na tento kód.

Příprava na SP

SP vede osoba odpovědná za bezpečnost produktu. Je vyžadována přítomnost vývoje, návrhu, testování, dokumentace a dokonce i projektového řízení. Důležité je zohlednit termín ohlášený zákazníkům. Informace o SP by měly být pro všechny zúčastněné dostupné na nějakém společném místě, většinou webových stránkách. Tyto informace zahrnují cíle, časový rozvrh a úkoly pro každého zúčastněného. Důležité je propojení se systémem pro sledování chyb (bugtracker) a mít aktualizovaný čítač chyb.

Důležité jsou také podmínky ukončení fáze SP. Podmínky mohou být následující:

- Všichni zaměstnanci mají splněné aktuální bezpečnostní školení.
- Veškerý zdrojový kód s vysokou prioritou byl zkontrolován.
- Model hrozeb byl zkontrolován a aktualizován. Pro všechny nové komponenty byl vytvořen nový model hrozeb.
- Byl zkontrolován AS a byl vymezen aktuální AS.
- Veškerá dokumentace byla z bezpečnostního hlediska zkontrolována.

Školení

I když většina týmu nebude potřebovat školení, vždy je nutno zvážit, zda školení před samotným SP nebude mít přínos. Obecně platí, že je dobré udělat alespoň menší školení o průběhu a postupu SP. Větší školení se vyplatí jen v případě, kdy se blíží expirace pravidelných celofiremních bezpečnostních školení.

Kontrola kódu

Při kontrole kódu se musí dbát na opravdu veškerý kód. Je nutno zkontrolovat jak nově napsaný kód, tak i ten starý. Dokonce je i důležitější více zkontrolovat starší konstrukce, jelikož spousta lidí bude ještě nadále po nějakou dobu používat starou verzi a jsou tak pro útočníka lukrativnější.

Prvním krokem kontroly je vytvořit databázi všech zdrojových souborů a každý přiřadit konkrétním osobám ke kontrole. Ke každému souboru by měla být vytvořena následující pole:

- Název souboru
- Majitel souboru (pokud není, tak ten kdo naposledy aktualizoval)
- Priorita (1 – 3)
- Kým zkontrolováno
- Zkontrolováno (s možnostmi: ano, ne, částečně)
- Poznámky

Dalším bodem je přiřazení priority. Ta se přiřazuje na základě toho, jaké programové konstrukce daný soubor obsahuje a odvíjí se to od modelu hrozeb. Dobrým přístupem při kontrole kódu je výměna kódu mezi dvěma programátory. Každý kontroluje kód toho druhého, čímž se zvýší pravděpodobnost odhalení chyby.

Důležitá je kontrola spouštěcích souborů. K takovým komponentám je vytvořena podobná tabulka jako výše a k testování je přiřazen tester. Ten testuje komponentu a nakonec doplní do tabulky, zda ji otestoval, jaké bezpečnostní testy provedl, zda je aktuální model hrozeb k dané komponentě a zda je AS aktuální.

Aktualizace modelu hrozeb

Během fáze SP je bezpodmínečně nutné, aby byly znovu zkontrolovány modely hrozeb. Tato poslední revize zaručuje, že jsou modely správné a kompletní. Dobré je zkontrolovat, zda některé DFD[11] není nutno změnit od poslední kontroly. Změny mezi etapou návrhu a etapou verifikace mohly tyto diagramy ovlivnit. Dále je potřeba se ujistit, že jsou všechny DFD elementy namapovány do STRIDE[12] kategorií. Nutná je aktualizace vstupních bodů aplikace a revize správného ukládání a uchovávání citlivých dat.

Bezpečnostní testování

I když se testováním zabývala předchozí fáze SDL procesu, zda je testování zaměřeno pouze na komponenty s nejvyšším bezpečnostním rizikem.

[11] Data Flow Diagram
[12] STRIDE = model pro zhodnocení bezpečnostních rizik(více v kapitole 3.1.6)

Vyčištění AS

Dalším krokem je kontrola AS, kde je nutno zamyslet se nad následujícími body:

- Spočítat všechny otevřené porty, sokety, SOAP rozhraní, RPC procedury a roury a rozhodnout se, zda jsou potřeba v základním nastavení.
- Ověřit všechny neautorizované vstupní body do sítě.
- Ověřit nastavení podsítí a nastavení bezpečných zdrojových adres.
- Zkontrolovat práva u jednotlivých procesů aplikace.
- Zkontrolovat ACL pro každý vytvořený objekt.
- Zkontrolovat naslouchání na jednotlivých portech.

Dalším přínosem kontroly AS je určení priority pro jednotlivé programové kontrukce. Testeři tak mají lépe určeno, kde je AS větší a kde je potřeba testovat více do hloubky.

Po této fázi je nutné opět aktualizovat AS dokument.

Vyčištění dokumentace

V této fázi by měli zaměstnanci určeni pro psaní a editování dokumentace ověřit všechny případy použití uvadené v dokumentaci, zkontrolovat, zda se neobjevují některé zastaralé údaje. Pokud se v dokumentaci objeví příklady kódu, měly by být ukázkou bezpečného případu užití.

Pokud v některém z případů užití může dojít k oslabení bezpečnosti aplikace, je nutné to nějakým způsobem zvýraznit. Například pokud nastavením otevřeme port pro neověřené uživatele, je dobré tuto informaci označit jako možné bezpečnostní riziko.

2.3.10 Fáze 9: Konečné zhodnocení bezpečnosti (FSR)

Tato fáze se zabývá otázkou, zda je produkt z hlediska bezpečnosti a soukromí připraven pro zákazníky. FSR kontroluje, zda bylo během celého vývojového procesu správně dodržováno SDL. Pokud tomu tak bylo, potom je fáze FSR jedna z nejkratších v celém SDL procesu. Fáze je rozdělena do čtyř podkroků, kterými jsou koordinace produktového týmu, shodnocení modelů hrozeb, shodnocení neopravených bezpečnostních chyb a ověření nástrojů.

Koordinace produktového týmu

Koordinace produktového týmu není technickou částí SDL. Jedná se spíše o fázi sběru obecných informací o projektu a ujištění, že jsou všechny věci tak, jak mají být. Musí být zapsáno, zda se jedná o samostatnou aplikaci nebo jde o nějaký druh balíčku (service/add-on pack) a zda aplikace komunikuje se sítí. Je nutné také uvést, kdy byl proveden SP a jak dlouho trval. Nutná je informace, kde jsou umístěny následující materiály:

- dokumentace Attack Surface
- modely hrozeb
- zdrojové kódy

- čítače chyb

K uvedeným modelům hrozeb je podstatné znát, kdy byly modely naposled zrevidovány. Důležité je se vrátit k požadavkům SDL a ověřit, že produkt všechny tyto požadavky splňuje. Pokud tomu tak není, je třeba ověřit proč. Poslední informací je záznam o běhu analyzujících nástrojů, konkrétně o jejich posledním běhu a výsledku.

Ověření modelu hrozeb

Ve fázi FSR je opět nutné zrevidovat všechny modely hrozeb. Důležitost se odvíjí od informace, kdy byly naposledy modely hrozeb aktualizovány. Čím starší datum, tím pravděpodobněji budou modely zastaralé.

Shodnocení neopravených bezpečnostních chyb

Úkolem tohoto kroku je zjistit, zda chyby které jsou na bugtrackeru označené jako neopravené jsou skutečně neopravené. Na tuto fázi by se měl vyhradit přesný čas a neměl by být překročen. Většinou by tato fáze neměla zabrat více než týden.

V systému pro sledování chyb by se měla vytvořit nová pole (SecAudit) pro chyby, které by odlišovaly důležitost opravy z hlediska termínu vydání. Potom je nutné všechny otevřené chyby projít a neaktuální chyby zavřít a aktuálním přiřadit správnou důležitost (severity).

Ověření nástrojů

Tento krok se zabývá analýzou použitých nástrojů při vývoji. Je kladen důraz na to, zda byly všechny nástroje správně spuštěny a nastaveny. U překladačů jsou zkontrolovány přepínače použité při překladu kódu.

Pokud byl proces SDL zaveden zodpovědně, potom by ověření nástrojů mělo proběhnout bez nalezení závažnějšího problému.

Co když se něco pokazí?

Obecně je známo, že pouze několik FSR proběhne na 100% bez problémů. Pokud je nalezena během FSR nějaká nesrovnalost, je nutné rozhodnout, jakou váhu má problém na produkt v očích zákazníka. Pokud například v kódu zůstalo několik zakázaných funkcí, pravděpodobně to nebude vadit. Pokud je nalezena závažnější chyba, potom je tato skutečnost diskutována s vedením a určí se, zda bude odloženo vydání nebo bude uvedeno v seznamu known issues daného vydání. Oprava je potom přislíbena do přístí verze (ve smyslu aktualizace Service Pack, neboli *dot release*[13]).

Obecně se ale tento proces rozhodování liší a v každé společnosti pro něj mají svoje jasně daná pravidla, obvykle uvedená v dokumentu Řízení změn.

2.3.11 Fáze 10: Plánování odezvy ohledně bezpečnosti

I když celý proces SDL je koncipovaný tak, aby nevznikaly bezpečnostní chyby, může se stát, že se u zákazníků objeví bezpečnostní zranitelnost a bude nutné na ni rychle reagovat. Není to dané tím, že by proces na tyto zranitelnosti nebral ohled, ale může se stát, že buď udělá vývojový tým nějakou chybu v implementaci nebo jednoduše při vývoji ještě taková zranitelnost neexistovala.

[13]dot release = tečkové vydání neboli například vydání verze 4.0.1 po verzi 4.0

Vždy když má být produkt vydán, nečeká se až se v produktu nenajdou žádné chyby, ale většinou se vydání odvíjí od předem stanoveného prahu, který udává počet nalezených chyb za dané období. Proto vždy zůstane v produktu jedna nebo dvě zranitelnosti.

Ať je proces SDL sebedokonalejší, nikdy nemůže počítat se zranitelnostmi, které ještě nebyly objeveny nebo které se ještě neprojevily. Jako jednoduchý příklad poslouží šifrovací algoritmy. V dnešní době jsou algoritmy jako SHA-1 a MD5 brány jako bezpečné, ale bude tomu tak i za 10 let, kdy kryptoanalýza pokročí a zároveň vzroste výpočetní výkon počítačů?

Příprava na odezvu

Prvním bodem odezvy je zřízení centra bezpečnostní odezvy (SRC). Je to entita, která bude řídit odezvu na nově nalezené zranitelnosti v produktu, který byl dodán zákazníkům. Proces začíná zaregistrováním problému, zjištěním, jak daný problém nastal a co může problém způsobit. Poté vývojový tým sestaví bezpečnostní záplatu, která vyjde v pravidelných aktualizacích produktu.

Centrum reaguje pouze na problémy, které se týkají verzí produktu vydaných směrem k zákazníkům. Pokud je zjištěna zranitelnost na vývojové verzi, tak je jednoduše opravena a do tohoto procesu není vůbec zapojena. Zde se je pouze potřeba ujistit, že byla zranitelnost odstraněna před vydáním nové verze. Centrum také reaguje na problémy, které byly reportovány vědeckými pracovníky nebo přímo útočníky. V takovém případě se zákazník nemusí o dané zranitelnosti dozvědět a je samozřejmě opravena v příští verzi.

Ještě je dobré se zamyslet, od koho vlastně zprávy o zranitelnosti chodí? Na tuto otázku není jednoznačná odpověď, ale vyjmenujme si tu několik skupin zabývajících se touto činností:

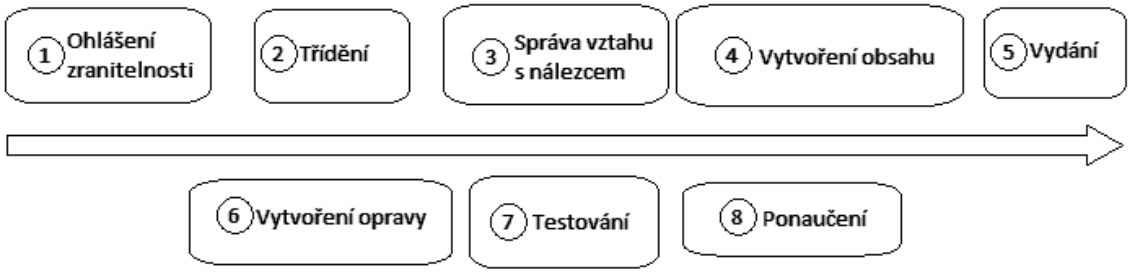
- výrobci bezpečnostních softwarů – naleznou tyto zranitelnosti a přídají do svých aplikací reporty o pokusech využití těchto slabin
- nezávislý bezpečnostní konzultanti – tito konzultanti prodávají informaci o zranitelnosti firmám zabývající se bezpečností nebo přímo výrobcům softwaru
- studenti IT – rozšiřují si své obzory v oblasti softwaru a bezpečnosti
- útočníci – buď přímo využívají těch slabin nebo informace o zranitelnostech prodávají kriminálním skupinám

Proces odezvy

Na obrázku 2.2 můžete vidět, jak probíhá odezva na bezpečnostní zranitelnost z hlediska centra bezpečnostní odezvy a z hlediska vývojového týmu.

Obrázek je rozdělen na dvě části. Vrchní část koresponduje s činnostmi, které vykonává SRC, kdežto spodní část obrázku popisuje akce provedené vývojovým týmem. Každá činnost – ať už SRC nebo vývojového týmu – je velice důležitá, a proto si zde o nich uvedeme více informací.

Ohlášení zranitelnosti je první částí procesu SRC. Většinou se pro ohlašování zranitelností používá e-mailová adresa společnosti s vhodně zvoleným aliasem. Centrum musí monitorovat všechny příchozí e-maily a zpracovat je. SRC by také mělo monitorovat diskuzní skupiny a fóra, kde se mohou objevit další informace. Tyto informace jsou většinou veřejné a proto je nutné reagovat rychle. Obecně platí, že by doba zpracování zprávy o zranitelnosti neměla trvat déle než 24 hodin (včetně svátků a víkendů). Pokud je zpráva



Obrázek 2.2: Reakce na bezpečnostní incident z pohledu SRC a vývojového týmu [5]

o zranitelnosti vyhodnocena jako relevantní, potom je zaslána zpráva o sdělení více informací reportérovi a také je vytvořen záznam o chybě v databázi SRC. Záznam je oznámen příslušnému produktovému týmu.

Třídění je proces k nalezení dostatečného množství informací o zranitelnosti a odhadnutí potencionálního dopadu na aplikaci. Nutno podotknout, že všechny příchozí hlášení musejí být roztříděny. Tým starající se o odezvu na hlášení se musí pokusit o reprodukci problému a pokusí se odhadnout případné útočníkovy úmysly. V konečné zprávě musejí být uvedeny všechny informace a vazby.

Vytvoření opravy je první akcí, kterou má na starosti vývojový tým. Pokud dojde k zařazení hlášení třídícím procesem, pak už je veškerá zodpovědnost o nápravě zranitelnosti na vývojovém týmu. Záplata opravující nahlášenou zranitelnost má 3 kritické aspekty:

1. Musí eliminovat nahlášenou zranitelnost.
2. Musí eliminovat všechny související zranitelnosti.
3. Nesmí narušit funkci kódu, kde byla zranitelnost nalezena.

Správa vztahu s nálezcem je velice důležitá část procesu hledání zranitelností. Pokud totiž nálezce opakovaně hlásí zranitelnosti, které jsou klíčové, výrobce si s ním vybuduje pracovní vztah a otevírají se nové možnosti spolupráce. Zároveň lze s takovým člověkem uzavřít dohodu, že slabé místo v aplikaci nezveřejní, dokud nebude chyba opravena v aktualizaci. Pokud je vybudován s nálezcem dobrý vztah, je mu umožněno dostávat v předstihu bezpečnostní záplatu pro další testování.

Testování bezpečnostních záplat má dva cíle:

1. Ověřit, že záplata skutečně opravuje danou zranitelnost nebo související zranitelnosti.
2. Pokusit se ověřit, že záplata nezpůsobí „krok zpět“ ve funkčnosti aplikace.

Testovací tým musí provést podrobné testy týkající se kódu, ve kterém byla nalezena zranitelnost. Musí se testovat všechny související vstupy aplikace, ne jen ty, kterých se přímo chyba týkala. Tým by měl mít také znalosti o hrozbách, které byly objeveny v konkurenčních aplikacích.

Vytváření obsahu slouží pro informování zákazníků o nalezené zranitelnosti. Dělí se na obsah pro profesionály a zákazníky. Profesionální obsah informuje IT profesionály

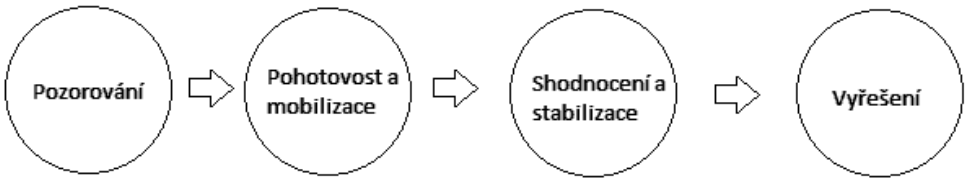
o zranitelnosti včetně potřebných technických informací. Zákaznický naproti tomu dodává informace pouze ve formě rad a upozorňuje uživatele na to, čeho by se měli vyvarovat. Vychází to z toho, že koncové uživatele většinou nezajímají technické dataily a chtějí být prostě chráněni. Na druhou stranu, pokud uživatele detaily zajímají, jsou jim k dispozici. Pokud se jedná o větší opravu (například o Service Pack), potom do této části procesu ještě spadá vytvoření obsahu pro média. Většina IT komunity sleduje elektronické kanály a opravy jako SP jsou velmi sledovanou událostí. Zároveň se informace o novém opravujícím balíku dostane do povědomí zákazníků nebo potencionálních zákazníků.

Vydání opravy nastane v momentě, kdy jsou všechny předchozí body splněny. Oprava je zveřejněna na veřejném místě a zabudována do systému automatických aktualizací (pokud takový systém aplikace podporuje). Stejně tak jsou vydány všechny informace spojené s danou opravou, případně zaslány prostřednictvím mailu odběratelům, kteří se k odběru těchto informací přihlásili. Ještě jsme se zde nezmínili o pravidelných opravách. Ty mají tu výhodu, že vycházejí v pravidelných intervalech a zákazníci tak s nimi počítají.

Ponaučení je poslední, ne však méně důležitou, fází vývojového týmu procesu odezvy. Ke každé zranitelnosti je vytvořen dokument, analyzující selhání v návrhu, programování, testování nebo školení, který popisuje, kde mohla nastat chyba, že se daná zranitelnost v produktu objevila. Je to vlastně část procesu, kdy se tým učí z vlastních chyb.

Proces okamžité odezvy

Jedná se o proces, který je spuštěn při nalezení nějaké velmi kritické zranitelnosti nebo když je pomocí zranitelnosti spuštěn červ, vir nebo jiný kus škodlivého kódu, který se rychle šíří. V praxi se tento proces označuje jako Software Security Incident Response Process (SSIRP). Cílem SSIRP je rychle mobilizovat zdroje společnosti a snažit se minimalizovat dopad na zákazníky používající produkt. Samotný proces můžeme vidět na obrázku 2.3.



Obrázek 2.3: Proces SSIRP [5]

Fáze pozorování začíná okamžitě, jak je některým týmem zaregistrováno neobvyklé chování. Obvykle SRC spouští tuto fázi, a to buď na základě vlastního pozorování nebo na základě příchozích hlášení o incidentech. Jakmile je incident potvrzen, přechází proces do druhé fáze.

Fáze pohotovosti a mobilizace nejdříve přidělí role účastníkům incidentu. Je sestaven tým SSIRP, krizový vedoucí, vedoucí inženýr a vedoucí komunikace. Produktový tým zodpovědný za postižený produkt je také svolán během této fáze. Ten začne zjišťovat technické detaily k incidentu a vedoucí komunikace začne zjišťovat dopady na zákazníky.

Fáze zhodnocení a stabilizace má za úkol informovat zákazníky, že hrozba byla zmírněna a že se intenzivně pracuje na opravě zranitelnosti, případně dodat informace o aktualizaci, které možnosti napadení zamezí. Důležitá je také komunikace s partnery a distributory o rozšíření informací o hrozbě ke koncovým zákazníkům.

Fáze vyřešení uzavírá incident tím, že jsou dodány nástroje, aktualizace nebo informace potřebné k zamezení efektů útoku a ochraně před možným dalším útokem. Pokud útok otevřel novou formu zranitelnosti a není na ní ještě záplata, potom je nutné vydat update ještě před tím, než je incident považován za vyřešený. V této fázi je také potřebná spolupráce s výrobci antivirových softwarů.

2.3.12 Fáze 11: Vydání produktu

Po celém vývojovém cyklu jsme dospěli do bodu, kdy je produkt hotový a připravený k vydání. Je potřeba zvolit vhodnou distribuční cestu, buď jako krabicový software (na CD nebo DVD nosiči) nebo formou stažení instalátoru z webu výrobce.

Bezpečnostní tým musí schválit, že byl během vývoje dodržen proces SDL. Většinou je tato kontrola dokončena s kladným výsledkem, protože bývají všechny problémy odhaleny během fáze FSR.

Během této fáze je také nutné, aby byly na firemní úložiště nahrány symboly k aktuální verzi sotfwaru. Symboly jsou dodatečná data pro debugger, na základě kterých je možno přeložit adresy a proměnné v přeloženém kódu do čitelné podoby. Tato data budou hojně využívat inženýři zabývající se podporou a laděním chyb přímo u zákazníků.

2.3.13 Fáze 12: Využití bezpečnostní odezvy

Poslední fáze souvisí s fází vytvoření plánu bezpečnostní odezvy (kapitola 2.3.11). Na základě dodržování zásad procesu SDL se očekává, že bude bezpečnostních incidentů co nejméně. Pokud však dojde k incidentu, je potřeba pečlivě prověřit všechny jeho okolnosti a zůstat v klidu a nepanikařit. Důležité je navázání kontaktu s člověkem (nebo skupinou lidí), který incident nahlásil. Důležitá je rychlá reakce, protože zákázníci jsou vždy na prvním místě. Tým zajišťující odezvu na incident by měl být vždy zdvořilý a snažit se s reportérem vytvořit dobrý vztah, díky čemuž bude dodáno více informací pro vývojový a testovací tým a chyba bude rychleji opravena.

Pokud si nejsme jisti všemi informacemi nebo kvalitou těchto informací, je nutno vyhradit pro incident více času a hlavně nic neuspěchat. Uspěchání může mít tyto následky:

- **Nekvalitní aktualizace** – aktualizace neopraví danou chybu nebo ji opraví pouze z části (zákazníci obecně nesnášejí, když musí instalovat více aktualizací opravujících jednu a tu samou chybu).
- **Reportér brzy po vydání aktualizace nalezne podobnou chybu** – tato situace nastane, pokud není incident řádně prošetřen.
- **Špatné otestování aktualizace** – aktualizace může narušit stabilitu aplikace nebo dokonce celého systému.

Dalším důležitým aspektem této fáze je sledování událostí, které mohou narušit vývojové plány. Je třeba také sledovat pokusy o napadení systému, případně postup dobývání systému. Na to přímo navazuje potencionální prolomení ochran, kdy je potřeba reagovat velmi rychle a chránit zákazníky před útočníky, kteří dané zranitelnosti využívají.

Asi nejvýznamnějším bodem této fáze je potřeba dodržovat plán a držet se postupů. Specifické postupy byly vytvořeny v nedávných fázích a je tedy potřeba neplýtvat zdroji.

Aby byl zajištěn rychlý postup práce, je nutné vytvořit seznam jmen a kontaktů, které budou v případě potřeby použity. Tento seznam je samozřejmě potřeba udržovat aktuální, případně ho doplnit o pracovní rozvrh jednotlivých členů, jelikož ne každý má stejnou pracovní dobu a v případě incidentu není třeba kontaktovat člověka, který v tu dobu nemá pracovní dobu, když je k dispozici člen, který má právě pracovní směnu.

Dále je třeba vedět, které kroky mohou být v případě krajní nouze přeskočeny. Někdy může nastat situace, kdy bude mnohem důležitější rychlá odezva než nutnost projít všemi kroky procesu. Přeskočení některých kroků je bráno v potaz tehdy, pokud je nalezena zranitelnost, která bývá okamžitě využívána útočníky a postihuje (nebo je pravděpodobné, že bude brzy postihovat) velké procento zákazníků. To potom lze zanedbat nutnost implementovat překlady do všech jazyků a verzí a stejně tak testování kompletní aktualizace do hloubky.

Kapitola 3

Analýza rizik

Tato kapitola se zabývá specifickou problematikou analýzy rizik, kterou je potřeba sestrojit při každém vývoji nového softwaru. Zvažují se zde rizika, která mohou nastat při samotném vývojovém cyklu, kdy největší dopady má většinou pozdní vydání softwaru.

Kapitola se nejdříve zabývá analýzou rizik podle procesu SDL. V procesu SDL je analýza rizik jednou specifickou fázi a v předchozím výkladu byla přeskočena.

Kapitola také částečně specifikuje požadavky pro analýzu rizik dle SDL a tyto požadavky jsou dále rozšířeny o znalosti analýzy pomocí chybových stromů. Shrnutí těchto požadavků je poté uvedeno na začátku kapitoly 4.

3.1 Analýza rizik dle SDL

Při tomto procesu je nutné dbát na to, aby tým provádějící analýzu rizik promyslel bezpečnost a soukromí aplikace ve všech směrech a do detailu. Analýza rizik se většinou provádí pomocí modelů hrozeb. Model hrozeb se skládá z několika částí:

- **scénáře použití** – konfigurace nasazení produktu v reálném prostředí
- **vnější vazby** – produkty, komponenty nebo služby systému, na kterých aplikace závisí
- **bezpečnostní předpoklady** – bezpečnostní služby nabízené ostatními komponentami
- **poznámky vnější bezpečnosti** – produktové info k bezpečnému ovládání aplikace

Výsledkem je dokument, který popisuje veškerá možná potencionální rizika. Dokument je součástí specifikace produktu. Tento model musí být aktualizovaný, kdy maximální stáří modelu by nemělo přesáhnout 6 měsíců.

Pro zamyšlení nad tím, co chceme vlastně modelovat, nám dobře poslouží následující přístup. Velké softwarové produkty jsou složeny z menších modulů, proto také modelování menších modulů aplikace je efektivnější než modelování celé aplikace. Aby se předešlo tomu, že produkt bude dobře zabezpečený na mikroúrovni, ale jako celek bude zranitelný, jsou mezi dílčími modely zavedeny tzv. *hranice důvěryhodnosti*. Všechny komponenty za jednou hranicí si mezi sebou důvěřují.

Tento model vytváří člověk, který má v oblasti bezpečnosti a soukromí nejvíce zkušeností. Model je součástí procesu návrhu skládajícího se ze tří částí. Nejdříve je to přípravná fáze, poté fáze analýzy a nakonec fáze návrhů na zmírnění následků rizik.

V následujících kapitolách si popíšeme fáze procesu tvorby modelu rizik.

3.1.1 Definice scénářů použití

Jak již vychází ze specifikace aplikace, tak je nutné vytvořit a definovat různé scénáře, jako například zcizení, uchovávání soukromých dat a ochrana před přístupem. Je potřeba také rozhodnout, na které klíčové scénáře hrozeb se zaměřit. To se přímo odvíjí od toho, jak bude produkt používán, zda bude mobilní nebo bude nainstalovaný na fyzických počítačích ve společnosti. Jednou z možností je také zohlednit, jaký zákazník bude produkt využívat, zda běžný uživatel nebo administrátor.

3.1.2 Vytvoření seznamu vnějších závislostí

Další částí je vytvoření seznamu všech souvislostí naší aplikace. Do této činnosti zahrneme například napojení aplikace na operační sytém (různé služby a vrstvy), napojení na databázi, internet, webové prohlížeče a další. V dnešní době je dobré i zohlednit různé balíky programovacích jazyků, jako například .NET Framework nebo common language runtime.

3.1.3 Definice bezpečnostních předpokladů

Tato fáze zahrnuje zvážení chování aplikace z hlediska bezpečnosti. Je třeba definovat úložiště, kde budou uchovány privátní klíče. Ty mohou být chráněny buď prostředky operačního systému nebo musí ochranu zajistit samotný produkt. Také sem spadá bezpečná komunikace na veřejné síti (většinou přes protokoly SSL/TLS) nebo zajištění exkluzivního přístupu k aplikaci samotné.

3.1.4 Vytvoření externích bezpečnostních poznámek

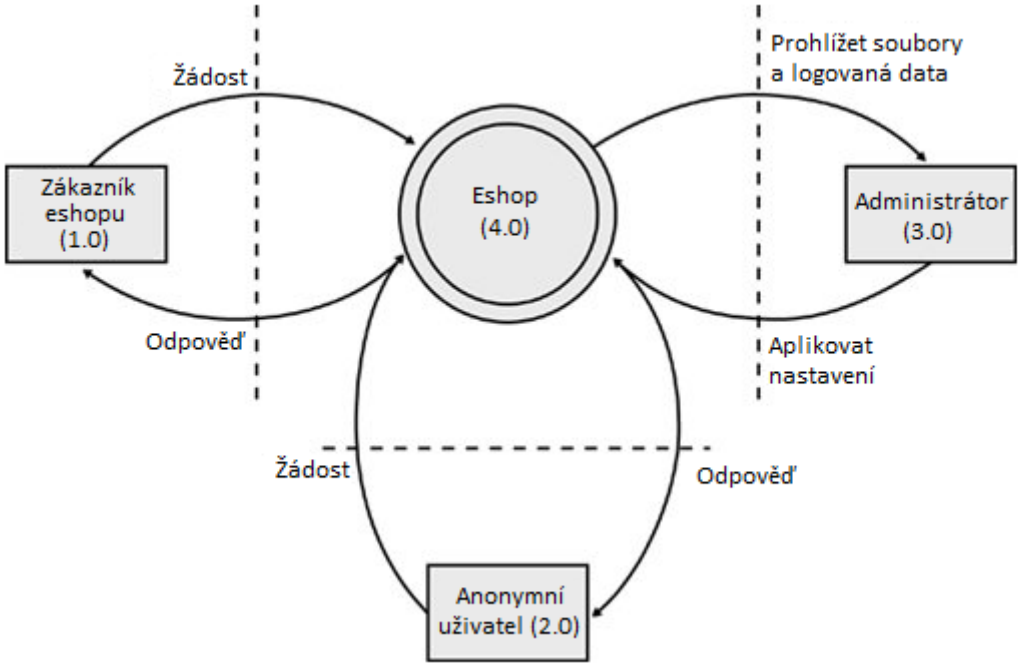
Tyto poznámky slouží výrobcům jiných aplikací, kteří chtějí komunikovat s naším produktem. Na základě poznámek budou rozumět bezpečnostním hranicím aplikace a jak ji správně a bezpečně používat. Tato fáze je proto velmi závislá na předchozích dvou fázích.

Při tvorbě poznámek je dobré se zamyslet nad tím, co je modelováno a na čem to závisí. Budeme vědět, co můžeme kontrolovat a tedy pro to i vytvářet model hrozby, nebo co nemůžeme kontrolovat, ale na něčem to závisí. Tato závislost má nějaké bezpečnostní předpoklady. Dobrým příkladem takového uvažování je tvorba aplikace na TCP/IP modelu. Ten je složen ze čtyř vrstev – aplikační, transportní, internetové a vrstvě síťového rozhraní. Při tvorbě modelu tedy musíme zohlednit vrstvy, které můžeme kontrolovat, čili většinou aplikační a někdy transportní. Oproti tomu vrtsvu internetovou a síťového rozhraní kontrolovat nemůžeme, ale můžeme pro ně (služby OS) vytvořit bezpečnostní předpoklady.

3.1.5 Vytvoření jednoho nebo více DFD

Účelem této fáze je vytvoření Data Flow Diagramů (DFD), které modelují celou aplikaci. Při návrhu je nesmírně nutné dbát na to, aby byl DFD správně navržený a korespondoval se skutečnou funkcí aplikace. Pokud je totiž DFD (nebo jeho část) špatně, potom je špatně celý proces.

Diagramy DFD jsou rozděleny na několik úrovní. Na nejvyšší úrovni je kontextový diagram, který vyjadřuje vztahy aplikace s uživateli, čili zobrazuje, kdo a jak k aplikaci přistupuje a jak na tyto akce aplikace reaguje. Jako příklad nám poslouží obrázek 3.1.



Obrázek 3.1: Kontextový diagram [5]

Na obrázku je možné vidět interakci různých účastníků se samotnou aplikací. Jak je vidět, kontextový diagram vyjadřuje aplikaci jako celek (soubor procesů) a dovoluje nám pochopit použití a akce.

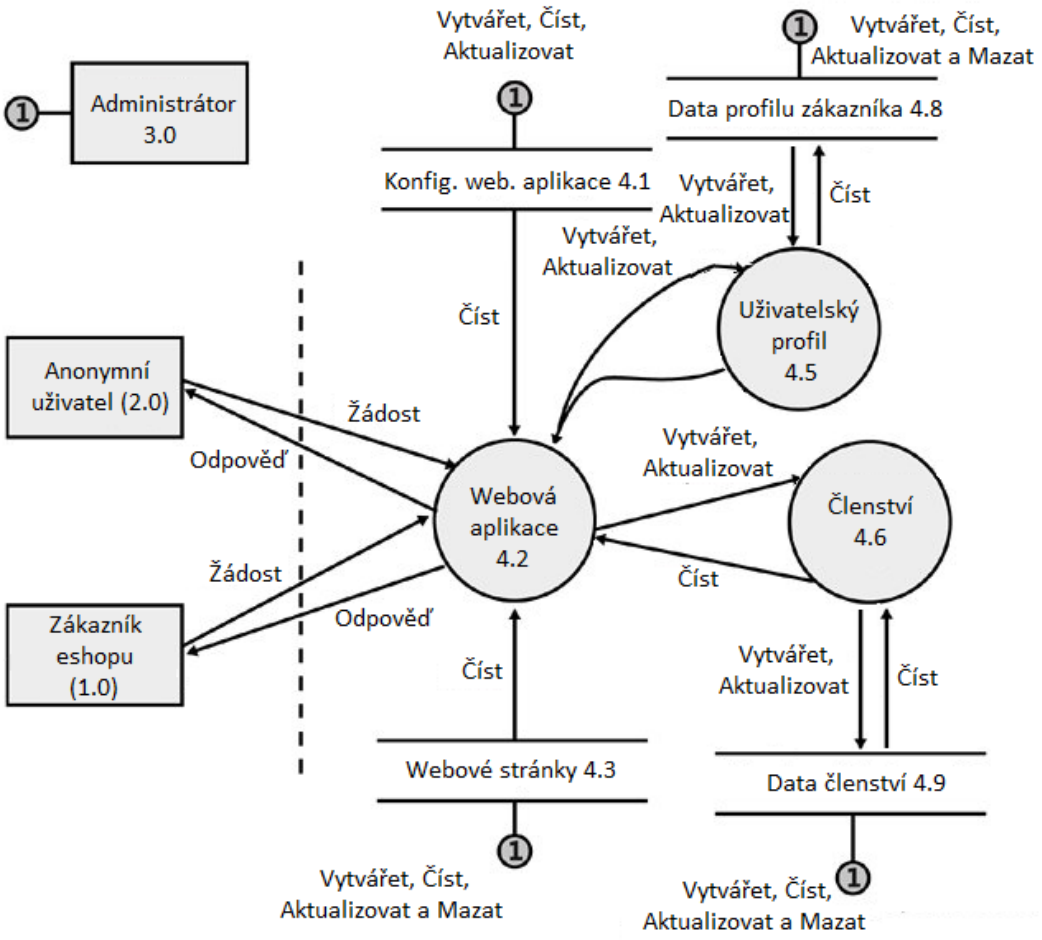
Dalšími grafy jsou diagramy označované jako level-x DFD, kde x vyjadřuje hloubku zanoření. Tyto diagramy popisují jednotlivé entity diagramů vyšší úrovně. Pokud se zaměříme na entitu *eshop* z kontextového diagramu na obrázku 3.1, potom by level-0 DFD mohl vypadat jako na obrázku 3.2.

Abychom ale lépe pochopili tyto diagramy, musíme si vysvětlit jednotlivé elementy DFD diagramů. Elementy jsou popsány v tabulce 3.1.

Dalším pravidlem při sestrojování DFD diagramů je číslování jednotlivých entit. Pokud je k DFD diagramu psán nějaký komentář, popisující čísla se vždy hodí ke správnému rozpoznání entity. U šipek vyjadřujících datový tok je zvykem doplňovat slovo, vysvětlující akci. V kontextu síťové komunikace se většinou používá dvojice slov *požadavek* a *odpověď*. Pro datové toky v kontextu souborů a databázových přístupů se používá terminologie CRUD[1]. Ta musí být ale většinou ještě doplněna informací, o jaká data se jedná.

Když se vrátíme zpět k obrázku 3.2, vidíme, že v diagramu již není element vyjadřující komplexní proces. To tedy znamená, že již není nutné vytvářet další DFD. Pokud by však DFD obsahoval komplexní proces, potom je nutné vytvořit další diagram, konkrétně level-1 DFD. Pokud v tomto DFD již komplexní proces nebude, tak se dále nezanořujeme. Také si v obrázku všimněme, že zanořené elementy dostaly číslování v hierarchické tečkové notaci pro lepší přehlednost.

[1]CRUD = terminologie označující akce se soubory a databázemi (create, read, update, delete)



Obrázek 3.2: Level-0 DFD [5]

3.1.6 Definice typů hrozeb

Pro definování typů hrozeb se používají různé modely. V souvislosti s následujícím textem si uvedeme 2 definice hrozeb.

Prvním je model CIA, který je založený na třech klíčových elementech a popisuje vlastnosti, které jsou z hlediska bezpečnosti žádoucí. Tyto vlastnosti jsou důvěrnost (Confidentaly), integrita (Integrity) a dostupnost (availability).

Druhým modelem je podrobnější a komplexnější model zvaný STRIDE. Tento model zvažuje hrozby z útočníkovy perspektivy. Každé písmeno názvu vyjadřuje možné ohrožení. Zde jsou vlastnosti tohoto modelu:

1. **S**poofing identity – krádež identity.
2. **T**ampering – narušení aplikace modifikací dat nebo kódu.
3. **R**epudiation – je v podstatě situace, kdy dojde k narušení logování a operace, které proběhly, nejsou zaznamenány.
4. **I**nformation disclosure – zpřístupnění informací neoprávněným osobám.

Tvar	Typ DFD elementu	Popis
dvojitý kruh	komplexní proces	Proces zastávající více operací. Jedná se o službu, démona nebo .exe proces s nalinkovanými dynamickými knihovnami.
kruh	proces	Proces, který vykonává jeden diskrétní úkon.
obdélník	externí entita	Někdo nebo něco spouštějící aplikaci, což ale aplikace nemůže kontrolovat. Mohou to být uživatelé, asynchronní události a externí procesy.
rovnoběžné čáry	datové úložiště	Trvalé datové úložište jako například soubory nebo databáze. Může také vyjadřovat data v cache.
šipka	datový tok	Zobrazuje tok dat v systému. Zahrnuje síťovou komunikaci, sdílenou paměť a volání funkcí.
čárkovaná čára	hranice důvěryhodnosti	Tato hranice je specifická pro modelování rizik a odděluje pohyb dat mezi oblastí s nízkou a vysokou důvěryhodností.

Tabulka 3.1: DFD elementy

5. **D**enial of Service – znepřístupnění služby.
6. **E**levation of priviledge – uživatelský proces využitím chyby získá administrátorská práva (root).

3.1.7 Identifikace hrozeb na systém

Jakmile máme hotové DFD diagramy, je potřeba vytvořit seznam všech DFD elementů. Každý tento element totiž musí být chráněn před potencionálním útokem. Seznam elementů potom bude vypadat dle tabulky 3.2.

V sekci Datové toky tabulky 3.2 můžeme vidět, že většina těchto akcí je obousměrná. Proto je vhodné zvážit redukci těchto entit. Taková redukce může být použita pouze na entity stejného typu, za stejnou hranicí důvěryhodnosti a pracující s podobnými daty. Na základě tohoto tedy může seznam redukovaných elementů vypadat jako v tabulce 3.3

Nyní když máme nejkratší možný seznam DFD elementů, tak můžeme aplikovat STRIDE na každý element v seznamu. Elementy lze zařadit do STRIDE kategorií podle převodní tabulky (3.4).

V podstatě jsou všechny typy DFD elementů náchylné k potencionálním útokům. Z tabulky můžeme například vyčíst, že *datové toky* jsou náchylné útokům typu modifikace datového toku (Tampering), odchytu informací z datového toku (Information Disclosure) a znepřístupnění služby (Denial of Service). V tabulce je také znak †, který je umístěný na průsečíku *datového úložiště* a útoku typu narušení logování. Toto vyjadřuje potencionální riziko, že pokud se útočník naboural do systému a modifikuje například nějaká data v databázi, potom může také zfalšovat logy a audity a zamezit odhalení průniku do systému.

Pro eshop nyní potřebujeme podle převodní tabulky určit jednotlivá rizika pro elementy

Typ DFD elementu	Číslo DFD entity
Externí entity	Zákazník eshopu (1.0) Anonymní zákazník (2.0) Administrátor (3.0)
Procesy	Webová aplikace (4.2) Profil zákazníka (4.5) Členství (4.6)
Datová úložiště	Konfigurace webové aplikace (4.1) Webové stránky (4.3) Data profilu zákazníka Data členství
Datové toky	Žádost anonymního zákazníka (2.0 → 4.2) Odpověď anonymního zákazníka (4.2 → 2.0) Žádost zákazníka eshopu (1.0 → 4.2) Odpověď zákazníka eshopu (4.2 → 1.0) Webová aplikace čte konfigurační data (4.1 → 4.2) Webová aplikace čte webové stránky (4.3 → 4.2) Administrátor vytváří nebo aktualizuje nastavení webové aplikace (3.0 → 4.1) Administrátor čte nastavení webové aplikace (4.1 → 3.0) Administrátor vytváří, aktualizuje nebo maže webové stránky (3.0 → 4.3) Administrátor čte webové stránky (4.3 → 3.0)

Tabulka 3.2: Seznam DFD elementů

daných DFD diagramů. Pokud budeme vycházet z předchozích tabulek a výsledků, tak konečné roztřídění hrozeb bude vypadat jako v tabulce 3.5.

3.1.8 Určení rizik

Dříve byly bezpečnostními specialisty rizika vyčíslována pomocí kalkulací. Problém s těmito kalkulacemi je, že mohou být velmi subjektivní. Hlavním vzorcem pro výpočet konkrétního rizika byl vztah:

$$R = p \cdot D$$

Výpočet popisuje, že hodnota rizika R je součin pravpěpodobnosti p, že daná událost nastane a čísla D, které vyjadřuje hodnotu potenciální škody. Systém sice funguje, ale jelikož nemůžeme předpovědět události přesně, potom je nutné počítat s odchylkami.

V Microsoftu to vyřešili jinak a implementovali do svých vývojových nástrojů jakýsi přehled chybovosti (v odborné literatuře dohledatelné pod pojmem *bug bar*). Ten vychází ze statistik jejich centra bezpečnostní odezvy (MSRC[2]). K indikaci celkové hrozby jsou pak použity různé úrovně nabývajících hodnot 1 až 4. Celkové riziko úrovně 1 je klasifikováno jako nejvyšší riziko, kdežto úroveň 4 označuje riziko nejnižší. Charakteristiky rizik zahrnují také různá hlediska a jsou popsána v tabulce 3.6.

[2]Microsoft Security Response Center

Typ DFD elementu	Číslo DFD entity
Externí entity	Zákazník eshopu (1.0) Anonymní zákazník (2.0) Administrátor (3.0)
Procesy	Webová aplikace (4.2) Profil zákazníka (4.5) Členství (4.6)
Datová úložiště	Konfigurace webové aplikace (4.1) Webové stránky (4.3) Data profilu zákazníka Data členství
Datové toky	Webová aplikace čte konfigurační data (4.1 → 4.2) Webová aplikace čte webové stránky (4.3 → 4.2) Žádost a odpověď anonymního zákazníka (2.0 → 4.2 → 2.0) Žádost a odpověď zákazníka eshopu (1.0 → 4.2 → 1.0) Administrátor vytváří, čte nebo aktualizuje nastavení webové aplikace (3.0 → 4.1 → 3.0) Administrátor vytváří, čte, aktualizuje nebo maže webové stránky (3.0 → 4.3 → 3.0)

Tabulka 3.3: Seznam redukovaných DFD elementů

Typ DFD elementu	S	T	R	I	D	E
Externí entita	X		X			
Datový tok		X		X	X	
Datové úložiště		X	†	X	X	
Proces	X	X	X	X	X	X

Tabulka 3.4: Převodní tabulka STRIDE pro typy elementů

3.1.9 Plánování zmírnění hrozeb

Fáze se zabývá redukováním nebo dokonce eliminací hrozeb potencionálně útočících na systém. Máme několik možností:

1. **Nedělat nic** – pro nízkorizikové hrozby.
2. **Odstranění dané funkčnosti** – riziko se redukuje na nulu. K tomuto kroku se svolí pouze při velmi vysokém riziku.
3. **Vypnutí dané funkčnosti** – pokud není funkce tak kritická, aby musela být odstraněna, stačí ji pouze vypnout.
4. **Varování uživatele** – uživatele vhodně upozornit na funkčnost s nízkou bezpečností.
5. **Proti hrozbě použít technologii** – například proti odposlouchávání použít šifrování.

Typ DFD elementu	Typ hrozby (STRIDE)	Číslo DFD entity
Externí entity	SR	(1.0), (2.0), (3.0)
Procesy	STRIDE	(4.2), (4.5), (4.6)
Datová úložiště	T(R)ID	(4.1), (4.3), (4.8), (4.9)
Datové toky	TID	(4.1 → 4.2), (4.3 → 4.2), (2.0 → 4.2 → 2.0), (1.0 → 4.2 → 1.0), (3.0 → 4.1 → 3.0), (3.0 → 4.3 → 3.0)

Tabulka 3.5: Konečné roztřídění hrozeb na jednotlivé DFD elementy

serverová aplikace	klientská aplikace
lokální přístupnost	vzdálená přístupnost
anonymní uživatel	autentizovaný uživatel
uživatelská přístupnost	administrátorská přístupnost
zapnuto v defaultním nastavení	vypnuto v defaultním nastavení

Tabulka 3.6: Hlediska charakteristik rizik

3.1.10 Použití modelu hrozeb pro kontrolu kódu

Nejdůležitější je nejprve začít kontrolovat kód, který je vzdáleně a anonymně přístupný. Tento kód je zranitelnější, jelikož k němu bude přistupovat více uživatelů. Teprve potom proběhne kontrola zbytku kódu.

3.1.11 Použití modelu hrozeb pro testování

Při testování je částečně potřeba vycházet z modelu hrozeb a zaměřit se na ty části kódu, kde by mohl být proveden nějaký útok. Je potřeba využít stromy hrozeb a testovat hlavně všechny hrozby na listových uzlech stromu.

3.1.12 Klíčové faktory úspěchu a metriky

Faktory úspěchu a měření, zda je model správně vytvořen není nijak jednoduché verifikovat. Napomoci nám k tomu mohou následující hodnocení kvality modelu.

0 – žádný model není – vyjadřuje, že nebyla pro daný produkt zvážena žádná rizika.

1 – nepřijatelný – buď je model zastaralý nebo je model starší než 12 měsíců.

2 – OK – DFD obsahuje entity, uživatele a hranice důvěrnosti. Pro každou entitu existuje minimálně jedna hrozba. Jsou zavedeny zmírnění hrozeb na úrovních 1, 2 a 3. Model je aktuální.

3 – dobrý – model splňuje požadavky 2 – OK. Na DFD jsou vyobrazeny anonymní, ověření, lokální a vzdálení uživatelé. Všechny hrozby typu STIE byly identifikovány a klasifikovány pro zmírnění nebo přijetí.

4 – výborný – model splňuje požadavky 3 – dobrý. Hrozby STRIDE byly identifikovány a mají navržené zmírnění. Pro každou hrozbu existuje zmírnění. Vnější bezpečnostní poznámky zahrnují plán pro vytvoření dokumentů pro zákazníky k bezpečnému použití aplikace.

3.2 Pravděpodobnostní zhodnocování rizik

Tato kapitola se zabývá pravděpodobnostním zhodnocováním rizik (PRA[3]). Nejdříve se dozvíme něco o tomto zhodnocování rizik obecně a poté přejdeme ke specifickému modelování této metody, kde se využívá analýza pomocí chybových stromů (FTA[4]).

3.2.1 Obecné informace

Tyto obecné informace byly primárně čerpány z [2]. PRA je systematický a komplexní přístup pro hodnocení rizik spojených s nějakým komplexním systémem. Jedná se o praktické techniky pro předpověď a správu rizik. Riziko je často charakterizováno pomocí dvou ukazatelů:

1. závažnost (severity) potencionálních nežádoucích účinků,
2. pravděpodobnost výskytu každého z následků.

Jednotlivé následky na systém jsou vyjadřovány numericky a možnost jejich výskytu je vyjadřována pomocí pravděpodobnosti nebo frekvence. Celkové riziko je poté vypočítáváno jako součet všech následků násobených dílčími pravděpodobnostmi, ale o tom až dále.

Metoda se používá při návrhu systémů, které musí zohledňovat velkou bezpečnost. Z běžného života si můžeme uvést návrhy jaderných elektráren, systémů vesmírných raketoplánů a raket, ověřování konstrukcí mostů a dálnic. Pokud bychom měli vybrat příklad z informační bezpečnosti, pak by byl vhodným kandidátem finanční sektor, konkrétně například webová aplikace internetového bankovnictví.

Výše uvedené návrhy většinou vykazují stejné nebo podobné vlastnosti, které je nutné zvažovat při analýze:

- Navrhovaný systém musí vydržet určitý stupeň zatížení a musí obsahovat různé záložní systémy a návrhy proti selhání.
- Správce rizik (nebo designér systému) rozhoduje o tom, jak se bude daný systém používat, čili má na starosti, jak se daný systém v různých situacích zachová.
- Prostředí systému je nejisté a generuje zatížení a nežádoucí podmínky, se kterými by se měl systém umět v ideálním případě vyrovnat.

Z dosavadních poznatků tedy víme, že je PRA používáno typicky na události velmi vzácné. Pro použití ale musíme znát údaje o systému. Vhodné je mít statistické údaje za delší časové období. Na základě těchto údajů pak lze jednoduše určit pravděpodobnosti jednotlivých událostí a zvyšuje se tak podobnost s daným systémem. Jelikož se však většinou jedná o úplně nový systém, je potřeba buď vycházet z nějakého příbuzného a podobného systému nebo se musí dané pravděpodobnosti odhadnout.

[3] Probabilistic Risk Assessment
[4] Fault Tree Analysis

Analýza rizik nám na základě hodnocení rizik poskytne informace k rozhodování při návrhu (bezpečnost, cena a výkon) a rozhodování při řízení (v jaké situaci daný systém vypnout).

PRA ještě v dnešní době není dokonalé, ale tato oblast má spoustu směrů, kterými se může dále vyvíjet. Pro ucelenost uveďme některé možnosti vývoje:

- Jak lépe určit chování systému a předpovědět pravděpodobný scénář na základě neadekvátních dat?
- Jak optimalizovat dílčí rozhodnutí, kterým čelí správce/operátor systému?
- Jak co nejlépe modelovat závislosti a nejistoty o aktuálním stavu systému?

3.2.2 Identifikace hrozby

Samotné PRA začíná definicí systému a analýzou a identifikací nežádoucích stavů, které se mohou za běhu systému vyskytnout. Identifikace hrozeb je vyvinuta pro identifikaci potencionálních nežádoucích účinků na běh systému. Pro identifikaci hrozeb se používají buď chybové stromy nebo stromy událostí. Tyto dvě metody si zde ve zkratce popíšeme a dále se budeme podrobně zabývat pouze metodou analýzy pomocí chybových stromů.

- **Analýza pomocí chybových stromů**
  - analýza začíná nežádoucím výsledkem (hlavní událost[5]) a důvody zpětně identifikující kombinace základních událostí, které vedly k vyvolání hlavní události,
  - výsledkem je strom, který obsahuje množiny základních událostí, které způsobí vyvolání hlavní události na základě propojených základních událostí pomocí AND a OR logiky,
  - strom se analyzuje od základních událostí, pro které známe pravděpodobnost buď na základě zkušeností, statistiky, odhadu nebo AND/OR dat.
- **Analýza pomocí stromu událostí**
  - začíná startovací událostí a postupně se propracovává k identifikaci potencionálních následků,
  - jedná se vlastně o rozhodovací strom ochuzený o rozhodovací uzly, čili je to pouze sekvence událostí s pravděpodobností na každé hraně podmínečně nezávislou na předchozí informaci,
  - pravděpodobnost (nebo frekvence) dané sekvence událostí je produktem daných hran po celé cestě průchodu stromem.

3.2.3 Analýza pomocí chybového stromu

Tato část má poměrně hodně kvalitních materiálů a pro čerpání znalostí jsem se rozhodl použít přednášku [3]. Analýza se v odborné literatuře vyskytuje pod zkratkou FTA (Fault Tree Analysis). Tato analýza identifikuje, modeluje a vyhodnocuje unikátní vnitřní vztahy událostí vedoucí k:

- selhání,

[5] v literatuře dohledatelné pod pojmem *root cause*

- nežádoucím událostem/stavům,
- nezamýšleným událostem/stavům.

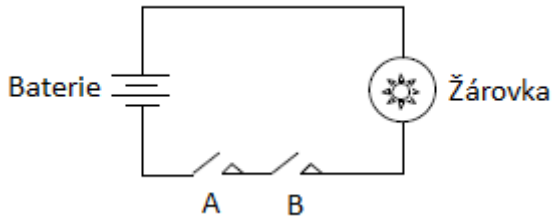
Analýza identifikuje na základě dedukce kořenové příčiny a poskytuje stanovení konkrétních rizik (částečné množiny, pravděpodobnost). Chybový strom poté vizualizuje dané závislosti a modeluje tak vztahy typu příčina – následek. Strom je postaven na pravděpodobnosti a obsahuje cesty, chybové události a normální události.

Metodika pro FTA je definovaná, stukturovaná a velmi propracovaná. Další výhodou je, že je velmi jednoduchá na naučení a poté i na používání. Využívá matematické a fyzikální jevy, jako například Boolovu algebru, teorii pravděpodobnosti, teorii spolehlivosti a logiku. Obecně vychází ze zákonů fyziky, chemie a konstruování.

Příklad jednoduché FTA

Ještě než přistoupíme k vysvětlování jednotlivých částí FTA, ukážeme si jeden velmi jednoduchý příklad FTA.

Mějme jednoduchý elektrický obvod, který obsahuje *baterii*, dva spínače *A* a *B*, *žárovku* a *propojovací vodiče*. Obvod vypadá jako na obrázku 3.3.



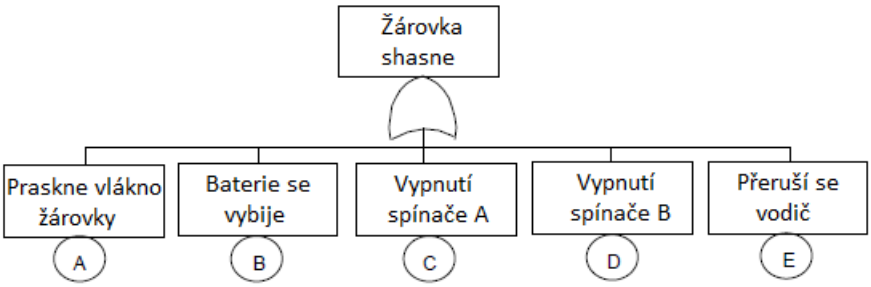
Obrázek 3.3: Jednoduchý obvod pro demonstraci FTA [3]

Abychom mohli udělat FT, potřebujeme zvolit nežádoucí událost. Zde je to poměrně jasné a nežádoucí událost bude **zhasnutí žárovky**. Nyní je třeba se zamyslet, které události mohou nežádoucí událost způsobit. Po zamyšlení dostaneme seznam událostí:

- vybije se baterie
- dojde k rozepnutí spínače A
- dojde k rozepnutí spínače B
- praskne vlákno žárovky
- přeruší se vodiče obvodu

Pokud se nad obvodem dále zamyslíme, uvědomíme si, že žádná z těchto událostí nesmí selhat a proto bude mezi událostmi vztah OR. Výsledný FT je na obrázku 3.4.

Ať nastane jakákoliv událost, tak dojde ke zhasnutí žárovky. Všechny dílčí události (A – E) tedy mohou způsobit nežádoucí událost. Ještě se zamysleme nad stavem, kdy by spínače *A* a *B* byly v paralelním zapojení. Potom by došlo k rozvětvení stromu a události C a D by musely být posunuty o úroveň níže. Události C a D by potom byly ve vztahu



Obrázek 3.4: Výsledný FT pro daný obvod [3]

AND (obě musejí selhat) a výstupem by byla událost například *oba spínače jsou rozpojené*, která by již byla ve vztahu OR k ostatním událostem. Zároveň by tím došlo ke snížení pravděpodobnosti vzniku nežádoucí události.

Použitelnost FTA

V této části si shrneme, jak se postupuje při analýze FTA.

Analýza nežádoucí události – je nutné identifikovat všechny relevantní události a podmínky vedoucí k nežádoucí události. Poté musíme určit paralelní a sekvenční kombinace událostí. Nakonec namodelujeme vnitřní vztahy komplexních událostí.

Zhodnocení rizik – na základě předchozího bodu (struktury stromu) spočítáme pravděpodobnost nežádoucí události (úroveň rizika). Identifikujeme bezpečnost kritické komponenty, funkce nebo fáze. Můžete také vyjádřit efektivnost změn v návrhu.

Zhodnocení bezpečnosti návrhu – demonstrujeme dodržení návrhu. Podle toho zdůrazníme, kde je potřeba zvýšit bezpečnostní požadavky. Identifikujeme a vyhodnotíme potencionální chyby v návrhu.

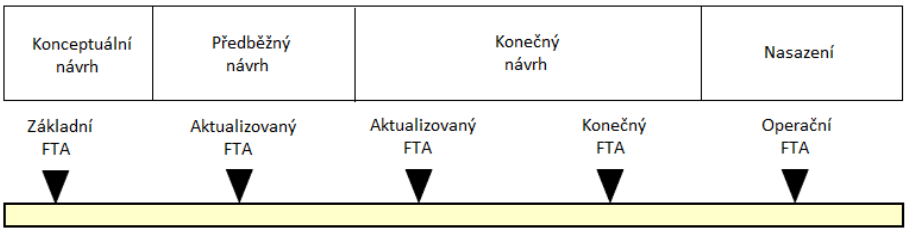
Časový rozvrh FTA

Časový rozvh FTA je rozdělen na dvě fáze. Nejdříve FTA postupuje svojí návrhovou fází a poté přejde do fáze operační. Obě fáze si bodově popišme:

Fáze návrhu – je nutné, aby tato fáze započala brzo během návrhu. Cílem je, aby se analýza zanesla do návrhu co nejdříve, protože pozdější změny jsou finančně nákladné. Během návrhu je nutné udržovat analýzu aktuální.

Operační fáze – během této fáze slouží FTA k analyzování nežádoucí události. V podstatě jde o hledání a řešení problémů v reálném čase.

Na obrázku 3.5 vidíme celý návrh rozdělený na několik částí (neustále se udržuje aktuální FTA) a poté i operační fázi.



Obrázek 3.5: Časový rozvrh FTA [3]

3.2.4 Stavební bloky

Pro konstrukci FT je nejprve nutné si představit jednotlivé stavební bloky. Bloků je více druhů a postupně si je zde všechny popíšeme. Dle kategorií můžeme stavební bloky rozdělit na následující:

- základní události,
- hradlové události,
- podmíněné události,
- přenosové události.

Základní události

Základní události vyjadřují základní stavy, které mohou na nejnižších vrstvách FT nastat. Dělíme je na chybové a normální události.

Chybové události jsou dvojího druhu. Prvním druhem jsou primární události (chybová událost, kdy dojde k chybě na základní komponentě), které se značí symbolem *kružnice*. Druhým typem jsou sekundární události (chybová událost, kdy je chyba způsobena externím zdrojem), které značíme symbolem *diamantu*. **Normální události** vyjadřují normální a očekávaný stav systému. Operace nebo funkce vyjádřená tímto blokem je zamýšlená a zanesená přímo v návrhu. Tento typ události je většinou dvoustavový a vyjadřuje stav zapnuto/vypnuto, což vyjadřuje převděpodobnost hodnoty 1/0. Normální událost se značí symbolem *obdélníku*.

Základní události jsou vstupními body pro ohodnocování bloků pravděpodobnostmi v FT.

Hradlové události

Hradlové události značí logické operace, které jsou prováděny na základě vstupních událostí. Na základě logického výpočtu hradlo povoluje nebo inhibuje chybovou logiku FT směrem nahoru a způsobuje další pokrok ve výpočtu FT. Máme pět základních typů hradel:

- AND,
- OR,

- inhibice,
- prioritní AND,
- exklusivní OR.

Podmíněné události

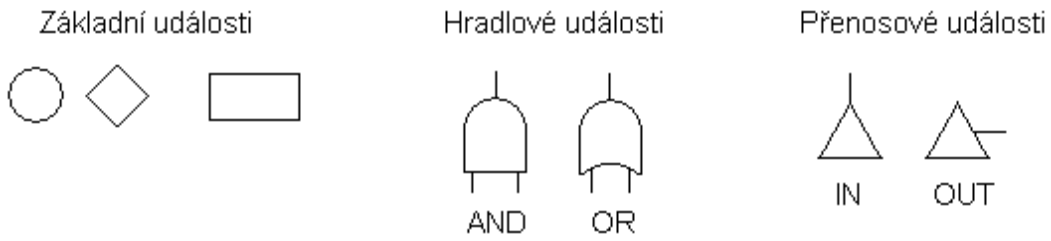
Tyto události v podstatě souvisí s hradlovými událostmi tak, že k danému hradlu přidávají podmínku pro logický výpočet. Řadíme sem události inhibice, prioritní AND a exklusivní OR.

Přenosové události

Tyto události slouží hlavně k zpřehlednění FT. Vyjadřují, že výsledek daného výpočtu je přenesen do jiné části FT. Dále také může indikovat, že výsledek z daného výpočtu je v FT použit na více místech. Přenosová událost je značena symbolem *trojúhelníku*.

Shrnutí bloků

Pro jednodušší představu o stavebních blocích se podívejte na obrázek 3.6.



Obrázek 3.6: Stavební bloky pro FTA [3]

3.2.5 Částečné množiny

V literatuře [3] se vyskytuje pojem *částečná množina* (cut set). Tento termín vyjadřuje množinu dílčích událostí, které vyvolají nežádoucí událost. Obecně existuje více než jedna částečná množina. S tímto pojmem souvisí další pojmy, které jsou ale pro další výklad nepodstatné.

3.2.6 Tvorba chybového stromu FTA

Samotná tvorba FT je rozdělena do několika úrovní detailu návrhu. Dělím je na *vrchní vrstvu* (tvar stromu, kombinace systémů), *střední vrstvu* (subsystémy, fáze, funkce a chybové stavy) a *spodní vrstvu* (základní události, komponenty chyb).

Pro samotnou konstrukci FT použijeme iterativní přístup. Ten je složen z několika kroků a postupně rozšiřuje strom od kořene k listům. Jednotlivé kroky jsou popsány následujícím seznamem:

1. Vyšetřit hradlovou událost.

2. Identifikovat všechny možné příčiny této události.
3. Ujistit se, že nebude přeskočena žádná možná příčina.
4. Identifikovat vztah a logiku událostí typu příčina – následek.
5. Doplnit strukturu stromu o tyto události a hradla.
6. Zpětně zkontrolovat, zda se události neopakují.
7. Zopakovat celý postup pro další hradlo.

Toto byl obecný postup, jak „naivním“ způsobem sestavit FT. My ale potřebujeme algoritmický přístup, aby byl detailní a vhodný pro implementaci. U toho postupu jsou uvedeny dvě zkratky INS[6] a PSC[7]. Vysvětlení plyne z názvů jednotlivých kroků. Algoritmický přístup je složen ze 3 kroků:

Krok 1 – Okamžité, Potřebné a Dostatečné (INS)

Identifikujeme všechny okamžité (nepřeskakovat minulé události), potřebné (zahrnout pouze to, co je skutečně potřebné) a dostatečné (nezahrnovat více událostí než minimální množství) události, vztahující se k vyšetřovanému stavu. Ověřujeme logiku, události a vztahy mezi nimi, dokud není nalezena nesrovnalost.

Krok 2 – Primární, Sekundární a Příkaz (PSC)

Na základě předchozího kroku pojmenujeme názvy událostí podle terminologie vstupu a výstupu. Zvážíme typ každé chyby pro spouštěcí události, což znamená rozhodnutí, zda je událost primární chyba, sekundární chyba nebo příkazová chyba. Na základě chyb/událostí vytvoříme hradlovou strukturu a identifikujeme události. Pokud se nejedná o základní událost, potom se jedná o příkazovou chybu, která je další spouštěcí událostí.

Krok 3 – Stav systému nebo komponenty

Nyní je nutno rozhodnout, jsou nově vzniklé entity stavy systému nebo stavy komponenty? Pokud se jedná o stav komponenty, potom je doplněna značka a zanořování neprobíhá. Pokud jde o stav systému, potom to znamená, že je takový stav složen z více událostí a je nutno iterativně postup opakovat.

[6] INS = Immediate, Necessary, Sufficient
[7] PSC = Primary, Secondary, Command

Kapitola 4

Návrh systému

Tato kapitola se zaměřuje na návrh produktu pro softwarovou podporu analýzy rizik dle SDL. Návrh plně vychází z výše uvedených informací a snaží se je správně a logicky sestavit do srozumitelného celku.

4.1 Spojení znalostí DFD, modelu STRIDE a FTA

V minulých kapitolách jsem popsal jednotlivé problémy týkající se DFD grafů, problematiku ohodnocování jednotlivých komponent DFD grafu pomocí převodní STRIDE tabulky a ohodnocování a strukturalizaci rizik pomocí FTA. Nyní je potřeba tyto znalosti spojit do jednoho celku a zajistit správnou spolupráci.

4.1.1 Základní kámen – DFD

Základním kamenem pro návrhovou část jsou správně navržené DFD. Správnost návrhu ale nebude součástí nástroje, tudíž sémantická správnost DFD musí být zajištěna externě. Navržený produkt by měl tedy zahrnovat pouze kontrolu na základní syntaktické úrovni. Pro tento návrh bude vhodné vytvořit nějaký typ pracovní plochy, kde bude možno přidávat a editovat jednotlivé komponenty grafu. Na pracovní plochu by mělo být tedy možné přidávat komponenty zobrazující procesy, datová úložiště, datové toky a externí entity. Systém by měl mít možnost ukládat informace o jednotlivých entitách a o vzájemném propojení.

4.1.2 Mezičlánek systému – identifikace hrozeb dle STRIDE

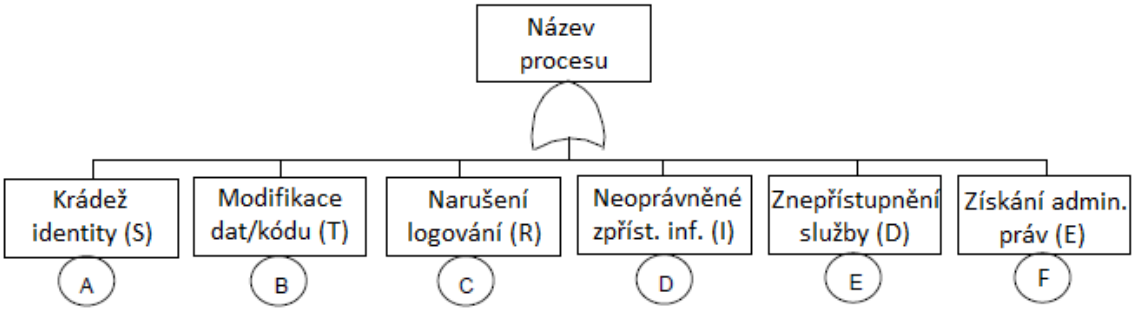
Pokud máme DFD diagram hotový, je potřeba přistoupit k přiřazení rizik podle STRIDE modelu. Rizika budou přidělována podle převodní tabulky 3.4. Ke každé entitě z předchozí podkapitoly tedy bude nutné ještě ukládat informace o možných hrozících rizicích.

Pro zhodnocení kvality modelu bude potřeba u každého rizika mít možnost přidávat informace o zmírnění daného rizika. Zhodnocení navíc umožňuje kontrolu na více typů zmírnění, proto bude potřeba tato zmírnění indexovat a umožnit zadání většího počtu takových informací k jednomu riziku.

4.1.3 Koncový článek – FTA

V tuto chvíli je vytvořený DFD diagram a jednotlivé entity DFD diagramu mají identifikovány hrozby pomocí modelu STRIDE. Nyní je potřeba nějak vhodně zahrnout tyto poznatky pro tvorbu FTA.

Každá entita má dle tabulky identifikovány hrozby, ale hrozeb je většinou více než jedna. FTA bude vytvářena ke každé entitě DFD. Riziko tedy bude bráno podle hrozeb korespondujících s modelem STRIDE pro konkrétní typ entity. Například pro proces hrozí všechny typy hrozby. Proto je nutné tyto hrozby nějak vhodně zakomponovat do celkového rizika. Celkové riziko tedy bude dáno jako hradlový typ OR přes všechna možná rizika u dané entity. Pro proces bude tedy vytvořen základ FTA stromu tak, jak ho můžeme vidět na obrázku 4.1.



Obrázek 4.1: Základ FTA pro entitu typu proces

Na obrázku je také vidět, že pod každým typem hrozby je kroužek s písmenem od A po F. Blok nad tímto kroužkem popisuje hrozbu, která bude sloužit jako kořenový prvek dílčího FT. Jde tedy o nežádoucí událost popisovanou v kapitole o FTA. Vhodné tedy bude zde mít odkaz na dílčí strom, aby byla zachována přehlednost.

4.2 Vyhodnocování

Předchozí kapitola popisovala zhotovení celé analýzy ve formě strukurovaných dat. Nyní bude potřeba vyhodnocovat celou analýzu. Hlavním prvkem vyhodnocování bude pravděpodobnostní údaj o daném riziku v daném bodě. Přepočítávání bude probíhat automaticky po doplnění relevantních informací pro FTA. Algoritmus pro výpočet bude uveden až v implementační části.

Druhým prvkem pro analýzu jsou různé ukazatele a metriky. Analýza dle SDL má definovány úrovně vyjadřující kvalitu modelu. Tyto údaje budou také zahrnuty do nástroje a budou vyhodnocovány v reálném čase. Jako doplněk by bylo vhodné, aby nástroj umožnil nápovědu pro zlepšení daného modelu. Například vezměme si případ, kdy má model úroveň 1 a obsahuje entity, uživatele a hranice důvěrnosti. Co už ale v modelu chybí je identifikace hrozeb. Aby mohl být model klasifikován na úrovni 2, je nutné mít pro každou entitu identifikovanou alespoň jednu hrozbu. Nápověda by teda měla uživateli říci, že musí doplnit alespoň jednu hrozbu ke každé entitě.

4.3 Návrh GUI aplikace

V této kapitole jsem se zaměřil na návrh grafického uživatelského rozhraní. Každou akci, kterou bude uživatel provádět, je potřeba implementovat intuitivně, aby uživatel musel co nejméně hledat v manuálu, jak danou věc provést. Aplikace bude koncipována jako

jednookenní, kdy bude možno přepínat kontext mezi DFD částí a FTA částí. Dále bude kladen důraz na to, aby byly vždy dostupné a dobře viditelné všechny potřebné informace.

4.3.1 Rozložení hlavního okna aplikace

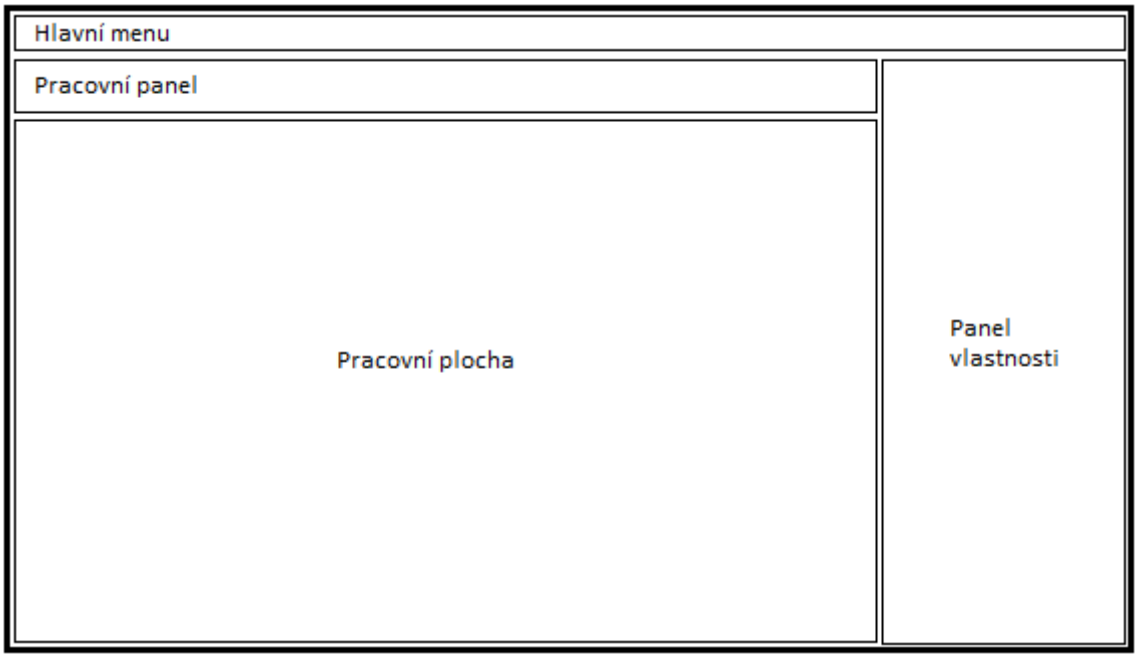
Hlavní okno jsem se rozhodl rozdělit na několik částí. Hlavní část bude vykreslovací plocha, která bude sloužit jako pracovní plocha pro náčrtek DFD diagramu a po přepnutí kontextu se na ní budou vykreslovat informace o konkrétní analýze, případně rovnou chybový strom dané FTA analýzy.

Nad pracovní plochou bude umístěna pracovní lišta, pomocí které bude možno spravovat kontext pracovní plochy. Na liště budou také přítomna tlačítka pro DFD náčrtek a pro FTA analýzu. U DFD náčrtku se bude jednat o tlačítka pro přidávání jednotlivých DFD prvků a u FTA analýzy to budou tlačítka pro přepínání mezi chybovými stromy. Na základě načtených dat v modelu budou některá tato tlačítka neaktivní díky absenci hrozby pro daný DFD prvek.

V pravé části hlavního okna bude panel s ovládacími prvky. Tyto prvky budou sloužit k zobrazování aktuálních informací o pracovní ploše. Budou zde také umístěna tlačítka pro práci s jednotlivými prvky. V případě DFD náčrtku zde bude tlačítko pro smazání daného prvku z náčrtku. V případě chybového stromu zde budou tlačítka pro přidávání a odebírání FTA prvků, aktualizaci výpočetních dat pro FTA analýzu a tlačítko pro zadání zmírnění hrozby.

V horní části okna nebude chybět ani hlavní menu, kde bude možno zadat nový model, aktuální model uložit nebo načíst již dříve vytvořený model. Budou zde také informace o nápovědě a o aplikaci.

Z výše popsaného textu tedy návrh hlavního okna aplikace vypadá jako na obrázku 4.2.



Obrázek 4.2: Návrh hlavního okna aplikace

4.3.2 Pracovní lišta

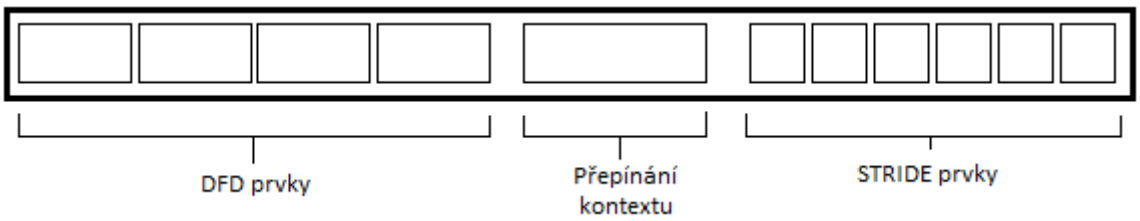
V předchozí sekci jsem uvedl přibližné rozložení okna aplikace. Nyní se podíváme, co bude na pracovní liště za ovládací prvky.

Lišta se bude skládat ze 3 skupin tlačítek. První skupinou budou přepínací tlačítka pro kreslení prvků do DFD náčrtku. Tato tlačítka jsou dvoustavová a vyjadřují, zda se má kreslit entita či ne. O tom se více rozepíši v návrhu pracovní plochy. Každé DFD tlačítko tedy lze aktivovat a deaktivovat. Aktivace tlačítka automaticky deaktivuje ostatní DFD tlačítka, proto může být vždy stisknuto buď právě jedno tlačítko nebo žádné tlačítko.

Druhou skupinou tlačítek je přepínání kontextu. Jedná se jednostavové tlačítko, kdy po kliknutí přejde aplikace ze stavu DFD návrhu do stavu FTA analýzy nebo naopak. Více o tomto přepínání bude uvedeno dále v této kapitole a poté také v kapitole 5.

Třetí skupina tlačítek slouží pro identifikaci hrozeb na DFD prvek vybraný v předchozím kroku. Tlačítka se chovají podobně jako skupina tlačítek pro DFD náčrtek. Poklikáním se zobrazí chybový strom pro danou hrozbu a v případě neoznačení žádného tlačítka budou vypsány obecné informace o analýze.

Detailní návrh pracovní lišty je tedy zobrazen na obrázku 4.3.



Obrázek 4.3: Návrh pracovní lišty

4.3.3 Panel vlastnosti

Tento panel hraje hlavní roli při zadávání a ohodnocování rizik. Pomocí tohoto panelu bude moci uživatel získat přehled o hrozících rizicích a případně je dále ohodnocovat a nechávat si pomocí dílčích rizik vytvářet analýzy pomocí chybových stromů. Panel je rozdělen na tři části.

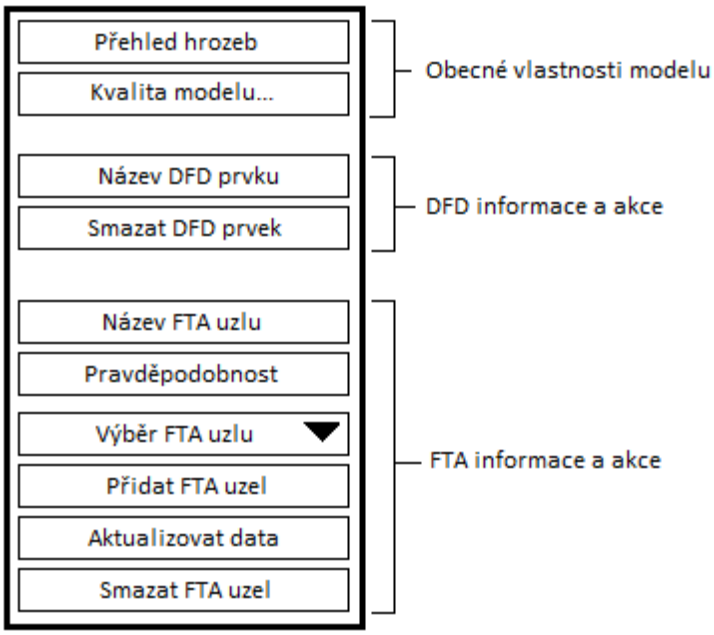
První částí jsou obecné informace o modelu. Na základě úplnosti informací zde bude tlačítko pro přehled o hrozících rizicích nad celým modelem, který bude sestaven do přehledné tabulky. Druhé tlačítko v této části bude uživatele informovat o kvalitě daného modelu. Více o kvalitě modelu na základě SDL jsme si řekli v kapitole 3.1.12.

Druhá část obsahuje informace o DFD, konkrétně popisek DFD prvku a tlačítko pro smazání DFD prvku. Popisek zde bude proto, že když budeme provádět FTA analýzu, tak na plátně nebude vykreslen DFD náčrtek a nevěděli bychom, pro který prvek je aktuální analýza prováděna. Tlačítko pro mazání bude aktivní pouze v prípadě kontextu DFD náčrtku a označeného některého DFD prvku.

Třetí sekce je nejdůležitější pro celou analýzu a práci s FTA stromy. Tato část obsahuje informace o FTA analýze, konkrétně o FTA uzlu, který je právě označený. Tato část bude obsahovat 2 textová pole s informacemi o názvu prvku a o ručně zadané pravděpodobnosti pro konkrétní FTA uzel. Dále se zde bude nacházet rolovací menu, pomocí kterého bude

možno označit FTA uzel, se kterým chceme pracovat. Na základě tohoto výběru budou aktivní tlačítka pro přidání a odebrání uzlu nebo pro aktualizaci dat u daného uzlu.

Návrh panelu vlastnosti je zobrazen na obrázku 4.4.



Obrázek 4.4: Návrh panelu vlastnosti

4.3.4 Pracovní plocha

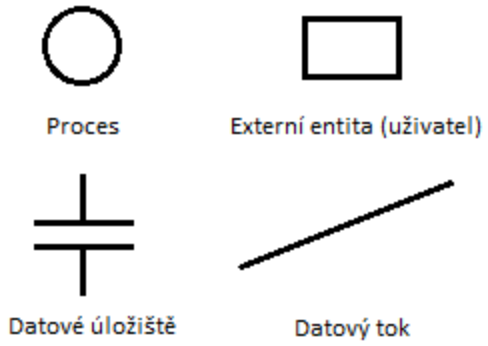
Jak jsem již v předchozím textu popsal, pracovní plocha bude mít přepínání kontextu a bude na ní tedy vykreslováno více typů dat. Pokud budeme v kontextu DFD náčrtku, potom se na pracovní plochu budou vykreslovat DFD primitivy.

DFD prvky jsou 4 různých typů. Prvním typem je proces, který označujeme symbolem kružnice. Druhým je datové úložiště, které je značeno speciální značkou, a to dvěmi rovnoběžnými úsečkami. Další je externí entita (někdy též označovaná jako *uživatel*) značená obdélníkem a poslední DFD entitou je datový tok, který je označován úsečkou mezi předchozími typy prvků. Jednotlivé DFD entity jsou na obrázku 4.5. Všechna tato označení vycházejí z knihy [5].

Pokud se budeme nacházet v kontextu FTA a bude vybrána nějaká analýza, potom budeme chtít vykreslovat FTA analýzu, konkrétně chybový strom. V různých publikacích se způsoby zakreslování stromů liší. Postupně jsem si prošel všechny způsoby vykreslování uvedené v materiálech [3, 4, 6, 7, 8] a nakonec jsem se rozhodl pro použití notace uvedené v [6]. Konečná podoba zvolená pro reprezentaci prvků chybového stromu je zobrazena na obrázku 4.6. Propojení mezi jednotlivými prvky bude řešeno pomocí kolmých spojnic.

4.3.5 Přehled hrozeb a úroveň modelu

Jednou z hlavních funkčností celé aplikace je zobrazení informací o daném modelu. Pro tyto informace, které mají pouze informační hodnotu a nemají možnost editace či doplňování



Obrázek 4.5: Vzhled jednotlivých DFD entit

Obrázek 4.6: Vzhled jednotlivých FTA prvků stromu

údajů, budeme používat informační dialogová okna.

Přehled hrozeb

Přehled hrozeb bude nejvhodnější zobrazit pomocí přehledné tabulky. Tabulka by měla mít dobře strukturované informace a měla by okamžitě informovat o hrozbách. Návrh, jak by tato tabulka mohla vypadat, můžeme vidět v tabulce 4.1, kde jsem jako příklad uvedl 2 DFD entity – eshop (proces) a databáze (datové úložiště).

DFD element	S	T	R	I	D	E	SUM (OR)
databáze	-	0.0025	0.0041	0.0011	n/a	-	n/a
eshop	n/a	0.001	n/a	0.00087	0.00036	n/a	n/a

Tabulka 4.1: Návrh vzhledu tabulky pro přehled hrozeb

V tabulce je vidět, že hrozby, které pro daný typ DFD elementu nehrozí, jsou proškrtnuté. Také jsou zde pole vyplněná řetězcem *n/a* značící, že pro danou hrozbu ještě nebyla hrozba spočítána (nebyly zadány všechny potřebné pravděpodobnostní údaje). Sloupec označený řetězcem *SUM (OR)* vyjadřuje celkovou pravděpodobnost napadnutelnosti daného DFD prvku. Celková pravděpodobnost je počítána jako OR nad všemi dílčími hrozbami, ale o tomto výpočtu se více zmíním až dále.

Úroveň modelu

Druhou informační hodnotou je úroveň modelu. Tu musíme hodnotit na základě zadaných údajů do modelu. Úroveň modelu zobrazíme jako dvě informační zprávy. První bude samotná úroveň (číslo od 0 do 4), která vychází z teoretického základu uvedeneného výše. Druhá zpráva bude uživateli říkat, co by měl na modelu vylepšit, aby se úroveň modelu zlepšila.

4.4 Návrh konkrétní vnitřní struktury dat

Jednou z nejdůležitějších věcí při tvorbě aplikace je správné navržení vnitřní struktury pro ukládání dat. Je potřeba zvolit vhodný přístup, aby bylo možné k těmto datům rychle a jednoduše přistupovat a využívat je k vnitřním výpočtům. Dále je nutné zajistit, aby tato data byla přístupná v těch částech aplikace, kde budou potřeba.

4.4.1 Hlavní datová část

Při startu aplikace bude vhodné, aby byla inicializována nějaká základní datová struktura, do které budou ukládána všechna ostatní data. Pro přístup k datům bude potřeba naprogramovat metody. Jako základní prvek tedy bude struktura vyjadřující model. Jak jsem již v předchozích kapitolách popsal, celkový model se bude skládat ze dvou spolu souvisejících částí. První bude část popisující DFD data a druhou část popisující FTA data. Tato data budeme mít od sebe oddělená, ale bude je možné identifikovat dle společného kontextu, ale o tom až dále. DFD část bude zastoupena pomocí seznamu DFD prvků a FTA část bude zastoupena pomocí seznamu korespondujících FTA analýz.

4.4.2 DFD část

Tato část modelu bude zahrnovat informaci o částech DFD modelu. Částmi modelu zde rozumíme DFD prvky – konkrétně to jsou entity: proces, datové úložiště, datový tok a externí entita. Jelikož jsou pro FTA analýzu prvky DFD pouze prostředníkem, není potřeba u nich z hlediska využitelnosti ukládat nějaké podrobnější informace. Pro naše účely tedy postačí, pokud budeme ukládat **název** DFD prvku a jeho **typ**.

Další informaci, kterou potřebujeme uložit, je **umístění** daného DFD prvku. Tyto informace nejsou využitelné při vyhodnocování modelu, avšak jsou o to důležitější při správném vykreslování DFD modelu aplikací. Na základě umístění DFD prvku budeme potřebovat ještě dodatečné informace o „naslouchací oblasti“. DFD graf chceme mít totiž interaktivní a chceme, abychom mohli tyto informace mít rychle k dispozici (a nemuseli je v reálném čase neustále přepočítávat).

Speciálním případem je DFD prvek typu datový tok. Tam je potřeba pro správné vykreslování ukládat 2 údaje o umístění (počátek a konec datového toku). S tím tedy souvisí doplnění DFD modelu o další proměnnou.

Poslední informací nutnou k uložení je nějaký mechanismus, jak bychom mohli spojit DFD prvek s pozdějšími FTA daty. Pro tyto účely jsem si zvolil doplnění dodatečné informace pomocí unikátního identifikátoru.

Z hlediska jednoduššího zpracování tedy bude DFD část modelu zastoupena prvky s následujícími informacemi:

- název DFD prvku

- typ prvku
- unikátní identifikátor
- 1. umístění prvku (všechny DFD prvky)
- 2. umístění prvku (pouze pro datový tok)
- 1. bod naslouchací oblasti (levý horní bod oblasti)
- 2. bod naslouchací oblasti (pravý dolní bod oblasti)

4.4.3 FTA část

V této části bude potřeba ukládat mnohem více informací než v části pro DFD model. FTA část, jak jsem již uvedl výše, bude zastoupena pomocí seznamu FTA analýz. Tento seznam bude tvořen prvky, kdy jeden prvek bude mít následující informace.

Prvními dvěmi souvisejícími informacemi jsou bezesporu unikátní identifikátor korespodující k DFD prvku a typ tohoto prvku. Tyto informace slouží pro vytvoření správného kontextu s DFD částí modelu. Další částí dat FTA analýzy jsou informace o šesti chybových stromech. Šest těchto stromů je nutno ukládat pro identifikaci všech hrozeb dle modelu STRIDE. Některé tyto stromy však nebude nutno použít, ale k tomu dojde až po inicializaci dle typu DFD prvku.

Každý chybový strom FTA analýzy reprezentuje výpočet dílčí STRIDE hrozby. Strukturu tohoto stromu je možné buď reprezentovat vnitřní datovou strukturou typu strom nebo strukturou typu seznam. Stromová struktura je složitější na implementaci, ale rychlost vyhodnocování je vyšší oproti struktuře založené na seznam. Po prostudování několika materiálů zabývajících se FTA analýzou jsem zjistil, že takové stromy málokdy nabývají větších rozsahů a proto jsem zvolil vnitřní reprezentaci pomocí seznamu. Pokud použijeme seznam, je také mnohem jednodušší spravovat unikátní identifikátory jednotlivých prvků (pomocí inkrementálního generátoru bez možnosti dekrementace).

Nyní jsme se dostali až k seznamu FTA prvků reprezentujících strom. Nyní se zamysleme, co je potřeba u takového prvku ukládat. Je určitě nutno uložit unikátní identifikátor FTA prvku. Jelikož se jedná o stromovou strukturu, je potřeba uložit informaci o rodičovském FTA prvku (s tím, že kořenový prvek takového rodiče nemá). Následovat budou FTA data potřebná pro počítání FTA analýzy. Stěžejní data pro výpočet jsou údaje pravděpodobnosti dané hrozby a logické operace u prvku. Doplňující informací je popis daného FTA prvku pro lepší orientaci.

Celková struktura FTA části modelu by tedy mohla vypadat následovně:

- unikátní identifikátor DFD prvku
- typ DFD prvku
- 6 chybových stromů obsahujících informace:
  - generátor unikátních identifikátorů FTA prvků
  - typ STRIDE hrozby
  - seznam FTA prvků obsahujících informace:
    - unikátní identifikátor FTA prvku
    - unikátní identifikátor rodičovského FTA prvku

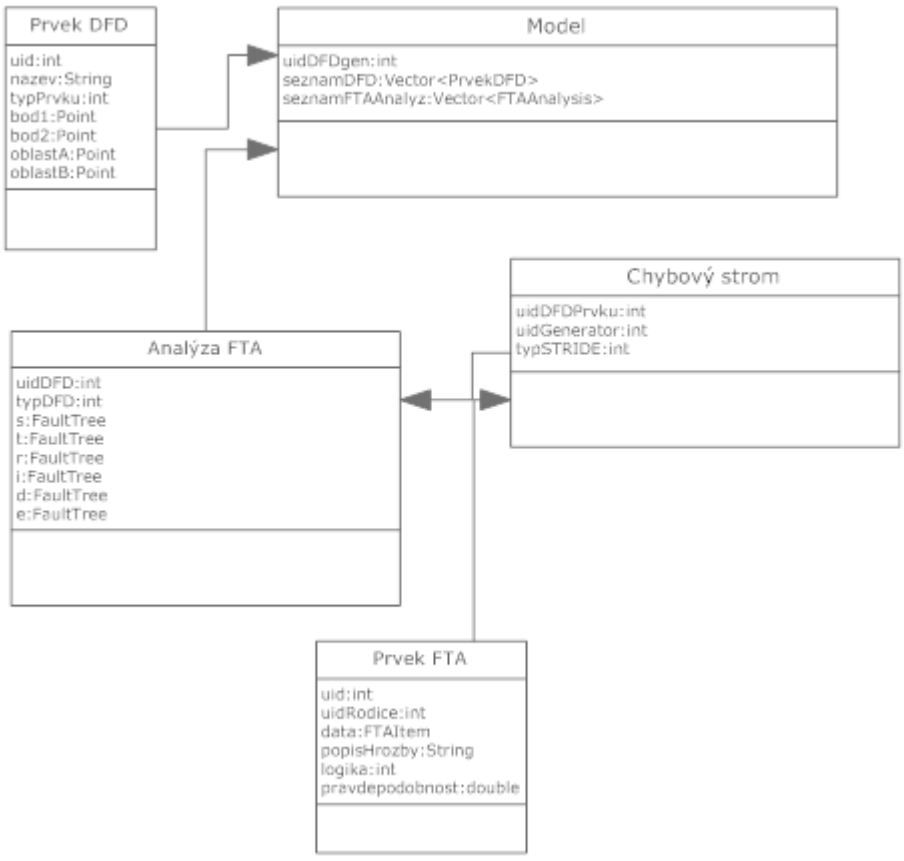
* popis FTA prvku
* pravděpodobnost hrozby
* logická operace (AND/OR)

4.4.4 Dodatek k hlavní části

Jak jsem popsal v podkapitole 4.4.2, bude u každého prvku potřebovat unikátní identifikátor. Toho jsem docílil tak, že jsem do kořenové struktury modelu přidal generátor těchto identifikátorů. Nejjednoduším přístupem byl generátor pracující na principu inkrementace (bez možnosti zpětné dekrementace v případě smazání prvku).

4.4.5 Návrh diagramu tříd

V tuto chvíli již tedy víme, jak bude vypadat vnitřní datová struktura pro ukládání všech informací potřebných pro práci s modelelem. Z těchto informací vychází návrh diagramu tříd. Zatím se jedná pouze o strukturalizaci dat a zatím chybí metody pro práci s těmito daty. Návrh digramu tříd můžete vidět na obrázku 4.7. Pokud by čtenáře zajímalo, jak byly skutečně datové třídy implementovány, potom v kapitole 5.4.6 uvádím přepracovaný diagram tříd podle toho, jak byl skutečně systém implementován, čili včetně metod.



Obrázek 4.7: Návrh diagramu tříd

4.5 Graf toku programu

Než jsem mohl přistoupit k implementaci jednotlivých funkcí dle návrhu, bylo potřeba ještě promyslet, jaké akce je možné v aplikaci udělat a odkud se kam můžeme dostat. Je to klíčové pro zpřístupnění všech relevantních informací na místech, kde mají být tyto informace dostupné a na druhou stranu blokovat zobrazení takových informací na místech, kde by neměly být viditelné nebo alespoň editovatelné.

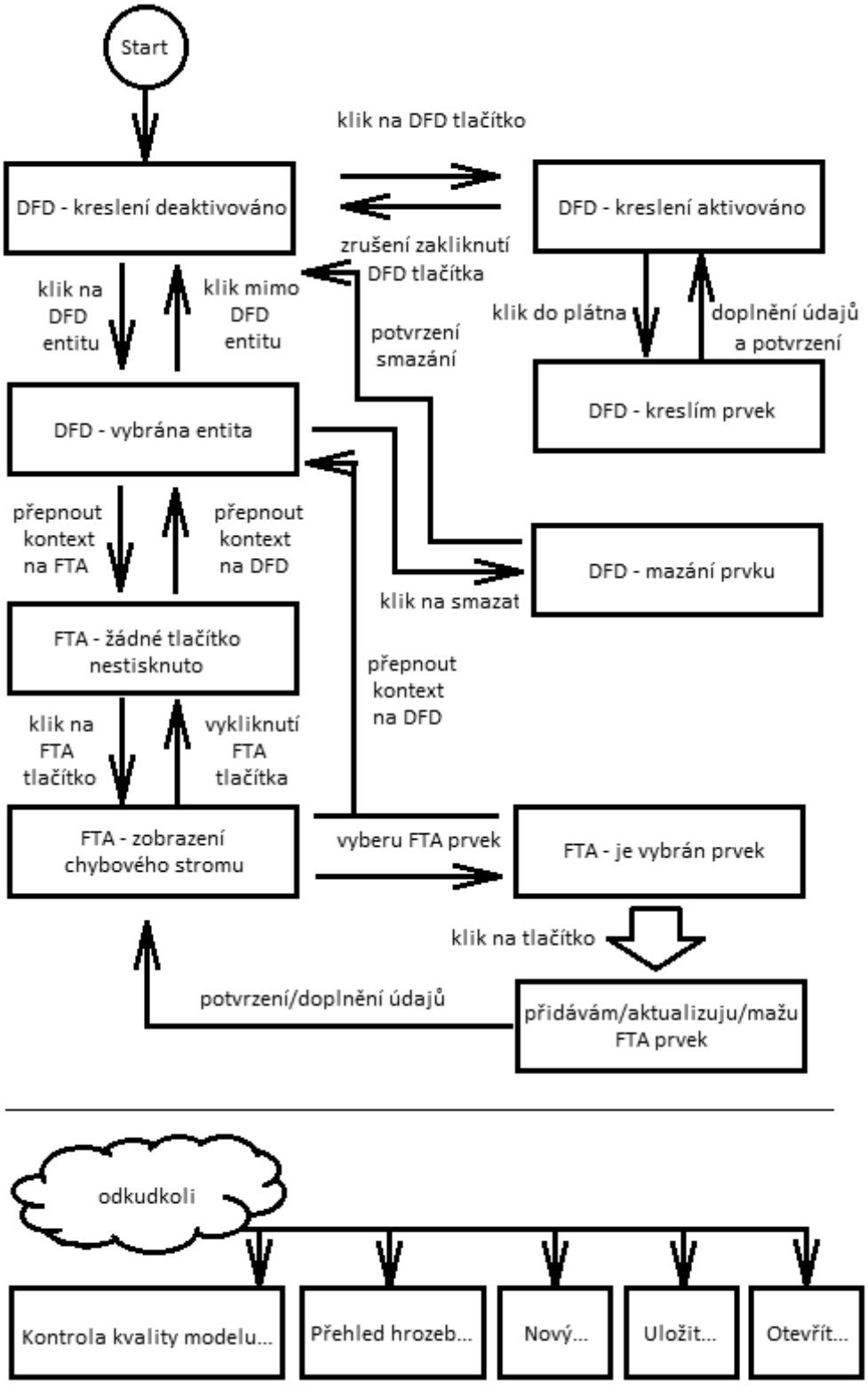
Na obrázku 4.8 můžeme vidět graf toku programu.

Z grafu vidíme, že po spuštění se aplikace dostane do jakéhosi počátečního stavu. V tomto stavu je kreslení na plochu deaktivováno a kontext celé aplikace se nachází ve stavu pro náčrtek data flow diagramu. Pokud tedy chceme kreslit diagram, musíme vybrat některé z tlačítek pro kreslení DFD prvku. Tlačítka jsou čtyři a po výběru můžeme kreslit dané entity. Výběrem některého prvku se dostaneme do stavu, kdy je kreslení aktivováno. V tomto stavu pracovní plocha naslouchá akcím myši. Pokud myší pohybujeme nad pracovní plochou, vykresluje se nám čárkovanou čarou obrys pro daný DFD prvek. Speciálním případem je kreslení datového toku, kdy po vybrání tohoto tlačítka musíme v grafu vybrat již existující DFD entitu a pomocí kliknutí se ukotví počáteční bod toku a druhým kliknutím se ukotví konečný bod toku. Takto můžeme kreslit diagram libovolného rozsahu. Pokud chceme kreslení DFD ukončit, vyklikneme tlačítko pro kreslení prvku. Nyní se nacházíme zpět ve stavu, kdy není kreslení aktivováno.

Najedeme-li nyní myší do náčrtku, poklikáním na jednotlivé prvky se nám tyto entity postupně označují. S označením entity se aktivuje tlačítko pro přepnutí kontextu, ale také tlačítko pro smazání daného DFD prvku. Pokud klikneme na tlačítko smazat, tak po potvrzení bude daný prvek smazán. Při vybraném prvku a po kliknutí na tlačítko pro přepnutí kontextu se dostaneme do stavu pro FTA analýzu. V tomto stavu se aktivují STRIDE tlačítka a naopak budou deaktivovaná tlačítka ohledně DFD. Zobrazí se přehled incidentů dle STRIDE, které mohou pro daný DFD prvek nastat. Pokud nyní klikneme na některé tlačítko ze skupiny STRIDE, potom se zobrazí chybový kořenový prvek daného STRIDE incidentu.

Při zobrazeném stromu můžeme data stromu aktualizovat. Pro práci s uzly stromu je nutné vybrat uzel, se kterým chceme pracovat. Pomocí roletového menu tedy vybereme uzel a klikneme na příslušné tlačítko. Je-li vybrán kořenový nebo nelistový prvek, tak můžeme provádět akce jako přidání nového poduzlu a aktualizace dat daného uzlu. Je-li vybrán listový uzel, potom můžeme ještě navíc daný uzel smazat. Provedeme-li některou z těchto akcí, potom se aplikace dostane zpět do stavu zobrazení chybového stromu pro daný DFD prvek a korespondující STRIDE incident.

Speciálním případem jsou tlačítka pro kontrolu kvality modelu, zobrazení přehledu hrozeb nad celým modelem a tlačítka z hlavního menu, pro tvorbu nového modelu, načtení již dříve vytvořeného modelu a uložení aktuálního modelu. Všechna tato tlačítka jsou dostupná a použitelná z kteréhokoli stavu aplikace. Jejich použití nijak nenaruší běh programu.



Obrázek 4.8: Graf toku programu

Kapitola 5

Implementace systému

Tato rozsáhlá kapitola se zabývá implementační částí celého nástroje. Z předchozích kapitol jsme se dozvěděli všechny informace, které budeme potřebovat k implementaci aplikace. Nejdříve budou představeny implementační nástroje, které byly pro vývoj aplikace použity. Následovat bude popis implementace grafického uživatelského rozhraní, základních mechanismů pro správné řízení toku programu, algoritmy pro výpočty jednotlivých modelů a stromových struktur, principy vykreslování těchto struktur a nakonec i implementace modulu pro importování a exportování modelů vytvořených naší aplikací.

5.1 Volba vývojových prostředků

Po zvažování, jaké implementační prostředí zvolit a v jakém programovacím jazyce aplikaci implementovat, jsem se rozhodl použít následující nástroje:

- Java Platform (JDK) 7 Update 2 (64bit) [1]
- Integrované vývojové prostředí NetBeans 7.1 [2]
- Knihovna pro práci s XML dokumenty Dom4j verze 1.6.1 [3]

5.2 Pomocné třídy a rozhraní

Ještě než přistoupím k samotnému popisu implementace celé aplikace, měli bychom si zde uvést informace o pomocných třídách. Tyto třídy byly vytvořeny pro zjednodušení a lepší orientaci při vývoji celé aplikace.

5.2.1 Třída Functions

Tato třída je implementována v souboru `Functions.java` a jsou v ní definované funkce pro ladění zdrojového kódu a správného chodu programu. Dále jsou v ní definovanané statické funkce pro získání rozměrů koncového zobrazovacího zařízení. Doplňkovou funkcí je převod konstant na textové řetězce vyjadřující popis daných objektů.

[1] volně ke stažení na: http://www.oracle.com/technetwork/java/javase/downloads/index.html
[2] volně ke stažení na: http://netbeans.org/downloads/index.html
[3] volně ke stažení na: http://dom4j.sourceforge.net/dom4j-1.6.1/download.html

5.2.2 Rozhraní Typy, Rozmery a FTA

Jedná se o rozhraní implementovaná pro přiřazení k jednotlivým potřebným třídám. Rozhraní `Typy` je použito při definování a rozpoznávání typů jednotlivých prvků DFD. Rozhraní `Rozmery` obsahuje informace pro správné přepočítávání rozměrů hlavního okna aplikace a rozhoduje také o správném rozložení prvků v tomto okně. Poslední rozhraní `FTA` je využívano při kreslení chybového stromu.

5.2.3 Třída Akce

Jedná se o nejdůležitější pomocnou třídu celé aplikace. Třída se nachází ve zdrojovém souboru `Akce.java` a obsahuje definici všech konstant používaných ve všech ostatních třídách či k řízení toku programu. Využití této třídy bude ještě zmiňováno v dalších kapitolách.

5.3 Implementace grafického rozhraní aplikace

V této kapitole popíši, jak bylo implementováno základní uživatelské rozhraní aplikace, tedy především hlavního okna. Implementace vychází z návrhu popsaného v kapitole 4.3. Jak již bylo v této kapitole popsáno, hlavní okno aplikace je rozdělené na několik částí, které budou impementovány odděleně. Tyto části však společně musí komunikovat a proto jsme se rozhodli, že je implementujeme v jedné společné třídě. Rozlišení jednotlivých částí potom bude pomocí metod, které pro ně budou definovány.

Hlavní třída celé aplikace je třída `RiskAnalysis`. Tato třída má jedinou funkci, a to vytvoření třídy reprezentující hlavní okno aplikace. Hlavní okno aplikace je reprezentováno třídou `MainApp`. V této třídě jsme si pro každou část hlavního okno vytvořili korespondující panel, na který bude vytvořena specifická část okna.

V konstruktoru této třídy tedy provedeme inicializaci panelů pro jednotlivé části okna, pomocí speciální metody přepočítáme rozměry a zavoláme postupně metody pro vytvoření jednotlivých částí grafického rozhraní aplikace. Postupně jsou volány následující metody:

- `vytvorMenu()`
- `vytvorPracovniPlochu()`
- `vytvorVlastnosti()`
- `vytvorToolbar()`

V následujících podkapitolách popíši implementaci jednotlivých částí okna aplikace. Než se však odebereme k jejich popisu, musím zmínit ještě jednu důležitou zanořenou třídu implementovanou uvnitř třídy `MainApp`. Je to třída `Udalost`. Tato třída dědí od standardní třídy `ActionListener` z vývojového balíku JDK. Třída má za úkol obsluhu jednotlivých akcí u tlačítek. Při vytváření tlačítka je potom přes metodu zavolána inicializace pomocí konstruktoru této třídy. Při volání konstruktoru se poté využívá již dříve popisovaná třída `Akce`. Pokud bychom tedy chtěli například přidat obsluhu u tlačítka pro smazání FTA prvku, potom by kód vypadal následovně:

```
JButton tlSmazFTAPrvek = new JButton("Smaž FTA prvek");
tlSmazFTAPrvek.addActionListener(new Udalost(Akce.SMAZ_FTA_PRVEK));
```

Tělo konstruktoru této třídy je tvořeno jedním velkým příkazem `switch`, kde je na základě přiložené proměnné rozhodnuto, jaká akce bude vykonána.

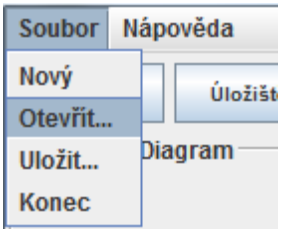
5.3.1 Vytvoření hlavního menu

Pro vytvoření hlavního menu byly použity standardní komponenty JDK. Byla vytvořena dvě oddělená menu.

První menu obsahuje nabídky pro vytvoření nového modelu, uložení aktuálního modelu vytvořeného aplikací a otevření staršího modelu. Pro obsluhu těchto akcí jsou v souboru `Akce` definovány konstanty, které jsou přidány pomocí třídy `Udalost` pro vykonání těchto akcí.

Druhé menu obsahuje položky pro nápovědu a má princip obsluhy stejný jako první menu.

Vytvořené menu je zobrazeno na obrázku 5.1.



Obrázek 5.1: Hlavní menu aplikace

5.3.2 Vytvoření pracovní lišty

Pracovní lišta je jedna z nejdůležitějších ovládacích prvků celé aplikace. Její tvorba je zajištěna pomocí funkce `vytvorToolbar()`. Invokací této metody se vytvoří dvě skupiny přepínacích (dvoustavových) tlačítek a jedno jednostavové tlačítko.

První skupina tlačítek vyjadřuje volbu DFD prvku a v jednu chvíli může být stisknuto pouze jedno z nich. Každé tlačítko má popisek jedné DFD entity.

Za první skupinou následuje jednostavové tlačítko, které přepíná kontext mezi kreslením DFD náčrtku a analýzou pomocí FTA. Toto tlačítko aktivuje a deaktivuje skupiny tlačítek pro DFD nebo FTA a to právě podle zadaného kontextu.

Za přepínacím tlačítkem následuje skupina dvoustavových tlačítek pro provádění analýzy podle FTA. Každé tlačítko nese název jedné hrozby dle STRIDE, které mohou pro daný DFD prvek nastat.

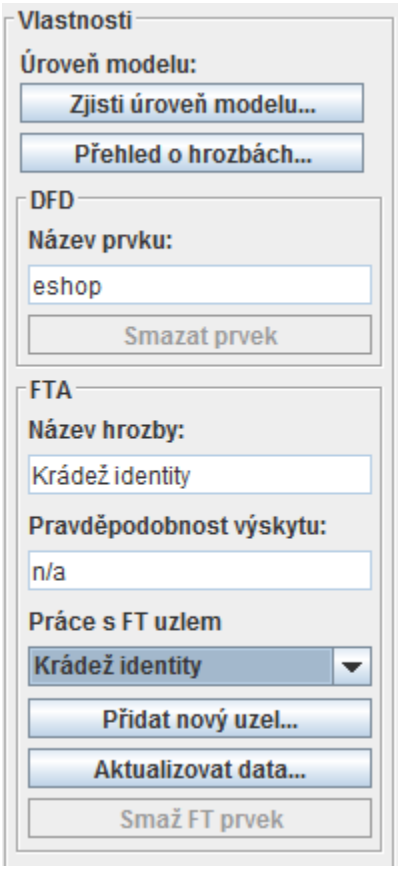
Obrázek 5.2: Pracovní lišta aplikace v kontextu DFD (1) a FTA (2)

Na obrázku 5.2 vidíme nejdřív vzhled lišty s kontextem pro DFD náčrtek, kdy jsou aktivní DFD tlačítka a FTA tlačítka jsou neaktivní a poté druhý náhled, kdy je naopak aktivní kontext FTA, aktivní jsou tlačítka pro FTA a DFD tlačítka jsou neaktivní.

5.3.3 Vytvoření panelu vlastnosti

Druhou nejdůležitější částí aplikace je panel vlastnosti. Díky tomuto panelu můžeme zadávat do modelu data a pracovat především s dílčími FTA analýzami. Implementace tohoto panelu přímo vychází z dříve uvedeného návrhu.

Panel je rozdělen na tři samostatné celky. Prvním je část obsahující 2 tlačítka pro zobrazení úrovně modelu a pro zobrazení přehledu hrozeb. Každé toto tlačítko má pomocí tříd `Udalost` a `Akce` napojeno reagování na stisk jednotlivých tlačítek. Druhý celek slouží pro práci s DFD prvkem. Celek obsahuje informaci o vybraném prvku a pomocí implementovaného tlačítka lze DFD prvek smazat. Obsluha stisku je opět součástí. Posledním, ale asi nejvíce důležitým celkem, je část pro práci s FTA stromem. K této funkčnosti jsou zde implementována textová pole pro zobrazování informací o daném uzlu, tlačítka pro práci s uzlem (Přidat, Smazat a Aktualizovat data) a poté roletové menu pro výběr daného prvku. K tlačítkům jsou doimplementovány akce pomocí již zmíněných tříd. Speciálním prvkem je zde roletové menu, o jehož obsluhu se stará speciální třída pojmenovaná `VyberPrvku`, kterou ale popíši až v kapitole 5.5.4.



Obrázek 5.3: Vzhled implementovaného panelu vlastnosti

Implementace takového menu můžeme vidět na obrázku 5.3, kde je aktuálně vybraný DFD prvek s názvem *eshop*. Kontext celého diagramu se nachází ve stavu FTA analýzy a proto je tlačítko Smazat prvek neaktivní. Aktuálně analyzovaná hrozba je *Krádež identity* a jelikož se jedná o kořenový prvek, tak je tlačítko Smaž FT prvek neaktivní.

5.3.4 Vytvoření pracovní plochy

Jelikož je pro uživatele nejdůležitější vizualizační část celé aplikace, tak jsem se rozhodl věnovat pracovní ploše většinu rozměru celého hlavního okna. Pracovní plocha je vytvořena jako velký panel ze standardní knihovny JDK. Na tomto panelu je umístěno plátno, jehož implementace je mnohonásobně komplexnější. O implementaci této části se stará třída `Platno` a je popsána dále v kapitole 5.5.

5.4 Implementace datových struktur pro model

Nyní, když jsme si popsali základní prvky grafického rozhraní aplikace, přejdeme k popisu datových struktur a tříd, které budou sloužit jako prostředník mezi jednotlivými funkcemi a pomocí jichž bude celý systém udržovat aktuální data buď načtená ze staršího modelu nebo aktuálně zadávaná uživatelem.

V návrhu (4.4.2 a 4.4.3) jsem uvedl, jak by taková struktura mohla vypadat. Během implementace došlo k minimálním změnám a bylo pouze doplněno zobecnění o jedno zanoření do struktury, ale o tom až dále.

5.4.1 Datová struktura Model

Jako hlavní strukturu celého datového systému jsem si definoval třídu nazvanou `Model` a implementovanou ve zdrojovém souboru `Model.java`. V celé aplikaci musí být zajištěno, aby existovala pouze jedna instance této třídy, která bude obsahovat veškerá data potřebná pro zobrazování a výpočty jednotlivých částí systému.

Kompozice této třídy je složena z několika proměnných a z metod, které s těmito proměnnými operují. Jelikož je celkový model složen ze dvou částí, z modelu DFD a z analýz FTA, je také tato vnitřní struktura rozdělena na dvě části.

První částí je DFD náčrtek, který je reprezentován seznamem DFD entit. Tento seznam jsme si pro tyto účely vytvořili jako vektor objektů třídy reprezentující jednotlivé entity DFD náčrtku.

Druhou částí jsou FTA analýzy. Tyto analýzy mají takovou vlastnost, že při existenci DFD prvku je zachována i existence příslušné analýzy. Analýzy jsou tedy reprezentovány opět vektorem, ale tentokrát vektorem objektů třídy reprezentující korespondující analýzu.

Třetí datovou částí je mechanismus, pomocí kterého je vytvářen kontext mezi DFD prvkem a jeho analýzou. Tímto prvkem je generátor unikátních identifikátů zajišťující jedinečnost daných prvků v modelu.

Z předchozích odstavců tedy vyplývá, že datová část třídy `Model` vypadá následovně:

```
public Vector<PrvekDFD> seznamDFD;              // prvky DFD náčrtku
public Vector<FTAAnalysis> seznamFTAAnalyz;     // korespondující FTA analýzy
public int uidDFDgen;                           // generátor unikátních id
```

Co se týče konstruktoru a metod implementovaných v této třídě, vždy je potřeba myslet na to, že kontext dvou výše definovaných vektorů musí být zachován. Pří volání konstruktoru jsou tady dané vektory inicializovány a generátor unikátních identifikátorů je nastaven na hodnotu 0. Kromě konstruktoru jsou zde definovány další metody:

- `pridejDFDprvek()`
- `smazDFDPrvek()`

- `vratPrvekDleUid()`
- `vratAnalyzuDleUid()`
- `vymazModel()`

Metoda `pridejDFDprvek()` zajistí, že se do vektoru DFD prvků přidá nový objekt reprezentující entitu a do vektoru analýz se vytvoří příslušná analýza. Je navíc zajištěno, že oba tyto typy objektů budou inicializovány s unikátním identifikátorem a na závěr bude vnitřní generátor inkrementován.

Metoda `smazDFDPrvek()` dělá přesně opačnou operaci. Smaže z obou vektorů prvek dle unikátního identifikátoru. U této metody však nedochází k zásahu do generátoru, aby byla zajištěna jedinečnost daných prvků.

Metody `vratPrvekDleUid()` a `vratAnalyzuDleUid()`, jak již název napovídá, vracejí dle zadaného unikátního identifikátoru objekt buď typu prvek DFD diagramu nebo objekt reprezentující analýzu.

K vymazání celého modelu (většinou při akci načítání starého modelu nebo při vytváření úplně nového modelu) slouží metoda `vymazModel()`. Metoda vymaže všechny prvky obou vektorů a nastaví generátor identifikátorů na hodnotu 0.

5.4.2 Datová struktura PrvekDFD

V předchozí podkapitole jsme u definice modelu narazili na vektor objektů typu `PrvekDFD`. Třída je definovaná ve zdrojovém souboru `PrvekDFD.java` a popisuje informace, které je nutné ukládat pro jednu entitu DFD náčrtku. Vyjdeme-li z návrhu v kapitole 4.4.2, je potřeba, aby tato struktura uchovávala 7 informací. Je třeba rozlišit, o jaký typ DFD prvku se jedná, jeho název, unikátní identifikátor pro korespondenci s FTA analýzou, 2 body pro umístění a 2 body pro naslouchací oblast. Konkrétně jsou informace implementovány takto.

```
int typPrvku;                  // typ DFD prvku dle třídy Akce
String nazev;                  // název DFD prvku
int uid;                       // unikátní identifikátor
Point bod1    = new Point();   // umístění entity
Point bod2    = new Point();   // u datového toku potřebujeme 2. bod
Point oblastA = new Point();   // levý horní roh naslouchací oblasti
Point oblastB = new Point();   // pravý dolní roh naslouchací oblasti
```

Jediná metoda implementovaná v této třídě je konstruktor. Jeho funkce spočívá v tom, že při vytvoření objektu známe pouze bod (v případě entity typu datový tok to jsou 2 body), kam se má prvek vykreslit, typ dané entity a název entity. Je nutno dopočítat oblast, kde má prvek naslouchat. O operaci naslouchání si toho více uvedeme v kapitole 5.5. Počítání oblasti je jednoduchou záležitostí pro entity typu proces, datové úložiště a externí entita, kdy je prostě určujeme podle jediného bodu a zadaných rozměrů grafické reprezentace dané entity. U datového toku však máme body 2 a musíme dbát na správnou orientaci toku. Mohou nastat 4 případy a tomuto výpočtu jsme nejprve implementovali složitou strukturu `if` a `else` větví. Po chvilku bádání jsme ale přišli na efektivnější řešení pomocí ternárních operátorů. Kód tohoto řešení vypadá takto, kdy vstupní body toku jsou `bod1` a `bod2` a výstupní spočítané oblasti jsou `oblastA` a `oblastB`.

```
oblastA.x = bod1.x < bod2.x ? bod1.x : bod2.x;
oblastA.y = bod1.y < bod2.y ? bod1.y : bod2.y;
oblastB.x = bod1.x > bod2.x ? bod1.x : bod2.x;
oblastB.y = bod1.y > bod2.y ? bod1.y : bod2.y;
```

5.4.3 Datová struktura FTAAnalysis

Nyní se dostáváme k druhé datové části modelu. Jedná se o analýzu FTA reprezentovanou třídou `FTAAnalysis` a implementovanou ve zdrojovém souboru `FTAAnalysis.java`. Tato analýza v sobě zapouzdřuje další složité struktury. Pro každý DFD prvek může totiž nastat dle dříve probírané teorie až 6 různých typů hrozeb. Pro každou tuto hrozbu je proto potřeba vytvořit 6 datových struktur reprezentujících analýzu konkrétní hrozby.

Kromě těchto struktur je vhodné ještě uchovávat informace o typu přidruženého DFD prvku a o jeho unikátním identifikátoru. Z tohoto důvodu jsem si zde vytvořil další dvě proměnné.

Datová část struktury `FTAAnalysis` tedy vypadá následovně:

```
public FaultTree s = null;
public FaultTree t = null;
public FaultTree r = null;
public FaultTree i = null;
public FaultTree d = null;
public FaultTree e = null;

public int uidDFD;
public int typDFD;
```

Jak je vidět v ukázce kódu, datové struktury `FaultTree` byly inicializovány na hodnotu `null`. Důvodem je implementace konstruktoru. Jelikož víme, že některé typy DFD prvků nemají potřebu identifikovat některé typy hrozeb, budou tyto nevyužité hrozby inicializovány právě hodnotou `null`. Samotný konstruktor je poté implementován jako větvený příkaz, kdy na základě vstupní hodnoty jsou inicializovány proměnné `uidDFD` a `typDFD`. Na základě druhé zmíněné proměnné jsou poté inicializovány pouze ty struktury reprezentující stromy, které odpovídají danému typu DFD prvku.

Kromě konstruktoru tato třída obsahuje ještě dvě metody. Jsou to metody sloužící pro určení kvality modelu, které se jmenují `jeHrozbaDefinovana()` a `jeNejakaHrozbaDefinovana()`. Využití obou metod bude podrobněji popsáno v kapitole 5.6.

5.4.4 Datová struktura FaultTree

V minulé kapitole jsem popsal strukturu popisující FTA analýzu. V této struktuře se vyskytovaly datové typy `FaultTree`. Tato část se bude zabývat podrobnějším popisem třídy, která tuto strukturu reprezentuje. Třída ja implementována ve zdrojovém souboru `FaultTree.java` a k této třídě je přidruženo již dříve popsané rozhraní `FTA`.

Datovou část třídy tvoří tři číselné proměnné. První proměnná reprezentuje unikátní identifikátor přidruženého DFD prvku, druhá proměnná slouží jako vnitřní generátor unikátních identifikátorů pro uzly chybového stromu a poslední obsahuje informaci o typu hrozby, kterou daný chybový strom analyzuje. Kromě těchto proměnných obsahuje datová část ještě seznam uzlů chybového stromu. Datová část tedy vypadá takto:

```
int uidDFDPrvku;        // ke kterému prvku se fault tree váže
int uidGenerator;       // generátor
int typSTRIDE;          // jakou hrozbu sleduje

List<PrvekFTA> seznam;  // seznam prvků stromu
```

Při vytvoření nové instance objektu reprezentujících strom musí zajistit správnou inicializaci vnitřních datových proměnných. Proměnné `uidDFDPrvku` a `typSTRIDE` jsou inicializovány na základě předaných parametrů při volání konstruktoru. Velmi důležité je správná inicializace generátoru unikátních identifikátorů. Tento generátor musíme inicializovat na hodnotu 0. V dále popsaných metodách pro manipulaci s uzly chybového stromu je tento generátor vždy pouze inkrementován. Poslední věcí potřebnou pro správnou inicializaci stromu je vytvoření seznamu prvků a vložení kořenového uzlu. Zinicializujeme seznam prvků, podle typu hrozby vytvoříme kořenový prvek a vložíme ho do nově vytvořeného seznamu reprezentující chybový strom. Například pro hrozbu *Znepřístupnění služby* by mohlo vypadat zinicializování seznamu prvků takto:

```
seznam = new ArrayList<>(); // vytvoříme seznam prvků
FTAItem data = new FTAItem();
data.logika = Akce.FTA_OR;
data.pravdepodobnost = -1;

switch(strideTyp)           // podle typu hrozby doplníme popis
{
    ...
    case Akce.D:
        data.popisHrozby = "Znepřístupnění služby";
        break;
    ...
}
PrvekFTA koren = new PrvekFTA(uidGenerator++, -1, data);
seznam.add(koren);  // pridame koren do seznamu
```

V kódu je možné vidět práci se zatím nepopsanými strukturami `PrvekFTA` a `FTAItem`. Tyto struktury budou popsány v rámci podkapitoly 5.4.5.

Abychom mohli s takto vytvořenou třídou dobře pracovat a provádět nad ní typické stromové operace, jsou v této třídě implementovány metody. Pro manipulaci s prvky jsou definovány následující metody:

```
public PrvekFTA koren()
public void pridejPrvek(int idOtce, FTAItem data)
public void aktualizujPrvek(int id, FTAItem novaData)
public void odeberPrvek(int id)
```

Metoda `koren()` vrací strukturu reprezentující kořenový prvek včetně informací. Metoda `pridejPrvek()` vytvoří nový prvek chybového stromu. Tento prvek bude mít inicializovány informace díky struktuře `data` a jako jeho rodičovský uzel bude nastaven prvek s identifikátorem podle proměnné `idOtce`. Pokud budeme chtít aktualizovat informace uzlu

chybového stromu, potom použijeme metodu `aktualizujPrvek()`, která na základě identifikátoru uzlu chybového stromu aktualizuje u dané položky nově zadané informace. Poslední manipulační metodou je metoda `odeberPrvek()`, která podle identifikátoru odstraní ze seznamu strukturu reprezentující uzel stromu.

Kromě těchto manipulačních metod je v této třídě implementováno několik metod napomáhajících při manipulaci se strukturou, ale mající spíše řídící charakter. Mezi tyto metody patří:

```
public boolean jeList(int uid)
public boolean maPotomka(PrvekFTA porovnavany)
public List<PrvekFTA> vratListy()
public List<PrvekFTA> vratPotomky(PrvekFTA rodic)
public PrvekFTA vratPrvekDleUID(int uid)
```

Pokud chceme u některého uzlu ověřit, zda se jedná o listový prvek, tak k tomu můžeme použít metodu `jeList()`. K této metodě máme definovanou inverzní metodu nazvanou `maPotomka()`, která ověřuje přítomnost potomka pro uzel zadaný identifikátorem. Potom zde máme definované dvě metody vracející seznam prvků. První – `vratListy()` – vrací seznam všech listových prvků daného chybového stromu. Druhá metoda s hlavičkou `vratPotomky()` na základě rodičovského prvku vrátí seznam prvků reprezentujících potomky daného rodičovského prvku. Poslední metoda nazvaná `vratPrvekDleUID()` vrací konkrétní uzel chybového stromu dle zadaného identifikátoru.

V této třídě jsou poté ještě implementovány dvě metody, které nejsou v aplikaci aktivně využívány. První je metoda `vypisStrom()`, která byla využívána pro ladící účely při vývoji, a to hlavně v situacích, kdy ještě nebyl implementován algoritmus pro vykreslování chybového stromu a bylo potřeba nějak kontrolovat správnost operací. Druhá je metoda `vratHloubku()`, která vrací maximalní hloubku chybového stromu. Při její implementaci bylo zamýšleno její využití při vykreslovacím algoritmu, ale nakonec však metoda nebyla použita.

5.4.5 Datová struktura PrvekFTA

V předchozí podkapitole jsem popsal strukturu popisující chybový strom. V této struktuře se vyskytoval seznam prvků, reprezentujících uzly chybového stromu. Tato podkapitola se zabývá implementací třídy `PrvekFTA`, která vyjadřuje právě uzly chybového stromu. Součástí této sekce bude také popis další třídy, která slouží pro zapouzdření konkrétních dat potřebných pro analýzu.

Třída `PrvekFTA` je datová třída. Neobsahuje žádné metody pro manipulaci s daty a jediným mechanismem implementovaným ve třídě je konstruktor pro vytvoření instance třídy. Konstruktor potřebujeme pro vytváření nových objektů, abychom mohli vkládat nová data a neukládaly se pouze odkazy na tato data.

V této třídě je potřeba ukládat tři typy informací. Prvním je unikátní identifikátor daného uzlu stromu, druhým je unikátní identifikátor rodičovského uzlu a třetím typem informací jsou data o daném uzlu, potřebná pro vyhodnocování analýzy. V návrhové části tohoto textu jsem si navrhl přímé zadání dat potřebných pro analýzu do tohoto typu třídy. Pokud však budeme myslet dále, je možné že se objeví nové možnosti analýzy a bude potřeba pro každý uzel ukládat více datových informací. Proto jsem se rozhodl datové proměnné zapouzdřit do další třídy nazvané `FTAItem`.

Z tohoto popisu tedy vyplývá, že třída reprezentující prvek bude obsahovat následující údaje:

```
int uid;            // unikátní identifikátor prvku
int uidRodice;      // identifikátor rodiče
FTAItem data;       // data FTA prvku
```

Zapouzdřená data uvnitř třídy `FTAItem` budou vypadat následovně:

```
String popisHrozby = "";
double pravdepodobnost = -1.0;
int logika;
```

Zapouzdřená třída tedy obsahuje informace o popisu hrozby, desetinné číslo vyjadřující pravděpodobnost s jakou daná hrozba může nastat a logickou operaci, která se bude aplikovat na všechna pravděpodobnostní data příchozí od uzlů potomků. Zapouzdřená třída dále nemá konstruktor, o její správnou inicializaci se staráme při vytváření nového prvku či při vytváření kořenového prvku u invokovaných metod třídy `FaultTree`.

5.4.6 Diagram datových tříd

Na základě předchozích podkapitol jsme tedy popsali datovou část aplikace. Abychom si lépe představili kontext, podívejme se na obrázek 5.4, kde je zobrazen diagram datových tříd aplikace. Narozdíl od diagramu uvedeném v kapitole o návrhu je tento diagram doplněný o všechny potřebné metody.

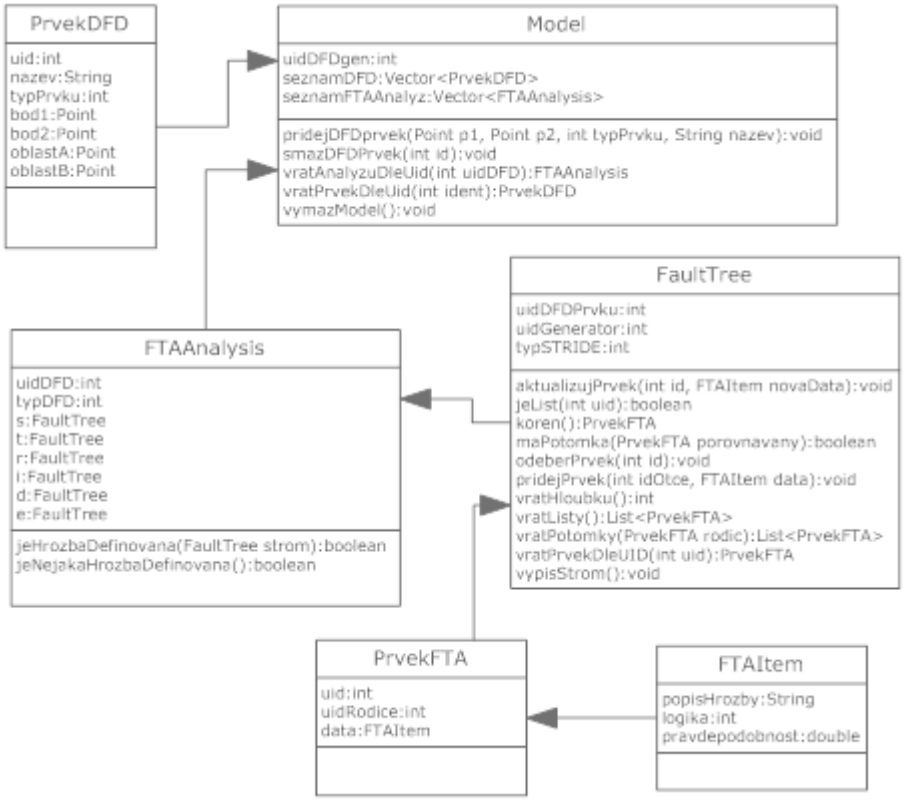
5.5 Implementace vykreslovací oblasti

Tato kapitola popisuje jednu z nejobsáhlejších implementačních částí celé naší aplikace. Jedná se o vykreslovací oblast. V kapitole popisující grafické uživatelské rozhraní jsme si uvedli inicializaci pracovní plochy. Byla vytvořena jako instance třídy `JPanel` a na ní byl vložen objekt naší třída `Platno`. Nyní si tedy popišme hlavní principy použité v této třídě.

Třída `Platno` byla vytvořena dědičností z třídy `Canvas`. Obecně je tato třída používána ke kreslení různých grafických primitiv, případně pro práci s rastrovou grafikou. Pro účely aplikace jsem si tedy odvodil vlastní třídu, kde jsem doimplementoval další metody.

V části konstruktoru bylo nutné tuto vykreslovací plochu nějak inicializovat. Bylo nutné zadat rozměry vykreslovací oblasti. Tyto rozměry (jak už bylo uvedeno v části zabývající se grafickým uživatelským rozhraním) byly předem propočítány a jsou tedy v době vzniku instance plátna známé. Pro účely jednoduššího přístupu k řídícím proměnným jsem do části konstruktoru ještě přidal odkaz na instanci třídy `MainApp`. Do konstruktoru bylo ještě nutné přidat inicializaci posluchačů událostí. V rámci třídy `Platno` jsem přidal dva typy posluchačů událostí. Bylo nutné naslouchat událostem myši, konkrétně události kliku tlačítka a události pohybu myši nad plátnem. Bohužel neexistuje třída implementující obě tyto akce a proto bylo nutné přidat posluchače dva. Pro tyto účely jsem pomocí dědičnosti definoval dvě třídy. Jejich hlavička vypadá takto:

```
private class KlikDoPlatna implements MouseListener
private class Pohyb implements MouseMotionListener
```



Obrázek 5.4: Diagram datových tříd

Třídy byly implementovány jako privátní zanořené třídy uvnitř třídy `Platno`.

Poslední akcí v konstruktoru je nastavení kurzoru myši. Kurzor je pro kreslící účely nastaven na znak terčíku pro určování konkrétní polohy kresleného prvku. Pro možnost výběru entity na plátně je poté aktivován kurzor ručičky.

Po zavolání konstruktoru máme tedy v rámci třídy `Platno` inicializovány následující proměnné:

```
// rozmery platna
private int sirka;
private int vyska;

public MainApp okno;         // odkaz na hlavní aplikaci
public Cursor kurzor;        // kurzor nad plátnem
```

Nyní když máme vykreslovací oblast inicializovanou, můžeme přejít k obecnému principu vykreslování. Vykreslování je zajištěno pomocí přepsané metody `paint()`. Uvnitř této metody máme kód rozdělený na dvě části pomocí řídící proměnné. Touto řídící proměnnou je promměná umístěná v hlavní třídě aplikace, konkrétně `okno.stav`. Proměnná vyjadřuje kontext, v jakém se právě aplikace nachází a může nabývat hodnot konstant `Akce.DFD` a `Akce.FTA`.

5.5.1 Princip vykreslování DFD náčrtku

Náčrt DFD grafu je výchozím bodem, pro který je poté počítána analýza pomocí chybových stromů. Dle návrhu jsem ho chtěl mít co nejvíce interaktivní, aby bylo možné rychle přejít k analýze hrozeb. Podle předchozí kapitoly jsem si tedy pro vykreslovací oblast definoval posluchače událostí myši, aby byla práce s náčrtkem rychlá a efektivní. Pokud nějaká událost nastane, potom je nastavena nějaká stavová proměnná (buď uvnitř třídy `Platno` nebo uvnitř hlavní třídy `MainApp`) a následně i souřadnicová proměnná a poté je zavolána metoda pro překreslení plátna. Než přejdeme k popisu principu vykreslování DFD náčrtku, podívejme se na implementaci jednotlivých posluchačů.

Pohyb – posluchač události pohybu myši

Tato třída nám naslouchá události pohybu myši a pokud tato událost nastane, tak nám vrátí instanci třídy `MouseEvent`. Z této instance je potom možné zjistit mnoho informací o události, nás ale především zajímají souřadnice nad plátnem, kde tato událost nastala. Toho využívám především při rozeznávání prvků na plátně. Máme totiž definované nějaké souřadnice kde víme, že se nacházejí objekty DFD náčrtku. Pokud nastane pohyb myší, tak pro souřadnici se vyhodnotí, zda tam není některý DFD objekt vykreslen. Pokud ano, tak se změní kurzor na ručičku (uživatelsky intuitivní pobídka ke kliknutí na daný prvek). Pokud tam není, je nastaven kurzor na terčík.

Speciálním případem je, když máme na liště zakliknuté nějaké tlačítko pro kreslení DFD prvku. Potom nepřepínáme kurzory, ale pomocí metody `kresliObjekt()` načrtáváme daný DFD prvek. K této metodě bude ještě v této kapitole podrobnější popis.

KlikDoPlatna – posluchač události kliknutí myši

Tato třída naslouchá událostí kliknutí myši a když tato situace nastane, vrátí nám opět instanci třídy `MouseEvent`. My opět získáme souřadnice. Kliknutí do plátna používáme pouze v kontextu DFD náčrtku a tam mohou nastat následující situace:

1. Není žádné tlačítko pro DFD prvek označeno.
2. Je nějaké tlačítko pro DFD prvek označeno:
    - Je označen typ datový tok.
    - Je označen typ externí entita, proces nebo datové úložiště.

Pokud není žádné tlačítko pro DFD prvek označeno, potom se jedná o naslouchání události, kdy hledáme na plátně entity k označení. V případě, kdy se trefíme do oblasti, kde se nějaký objekt nachází (vlastní metoda `kteryDFDbylOznacen()`), potom je nastavena vnitřní proměnná `uidZvyraznenyDFD`. Zároveň se aktivují tlačítka pro smazání DFD prvku a pro přepnutí do FTA analýzy (je aktivována práce s daným DFD prvkem).

Pokud je některé tlačítko pro DFD prvek označeno, potom mohou nastat 2 případy. Když bude označeno některé z tlačítek *proces*, *externí entita* nebo *datové úložiště*, potom stačí na základě údajů o pozici, typu označení a popisu prvku uložit daný prvek do seznamu DFD objektů. Pokud však bude označeno tlačítko pro *datový tok*, potom je nutné získat 2 body na plátně (počátek a konec datového toku). Ty navíc musí být umístěny v naslouchacích oblastech již přítomných DFD prvků v náčrtku. K tomu mám definovány pomocné proměnné pro identifikaci, zda se již kreslí datový tok, či zda se čeká na označení počátku

toku. Ať už se kreslí datový tok či ostatní prvky, vždy když jsou k dispozici všechny údaje o pozici DFD prvku, tak je zavolána metoda pro zjištění jeho názvu. Jedná se o metodu `zeptejSeNaText()`, která uživateli zobrazí dialog pro zadání názvu prvku. Po zadání názvu je prvek vložen do seznamu vykreslovaných DFD prvků. S použitím metody uvedené v předchozích kapitolách vypadá přidání prvku do seznamu vykreslovaných následovně:

```
...
// vykreslování datového toku
okno.model.pridejDFDprvek(new Point(pocatek), new Point(x, y),
                          Akce.DATA_FLOW, text);
...
// vykreslování ostatních prvků
okno.model.pridejDFDprvek(new Point(x, y), new Point(),
                          okno.typOznaceni, text);
...
```

Princip vykreslování DFD náčrtku

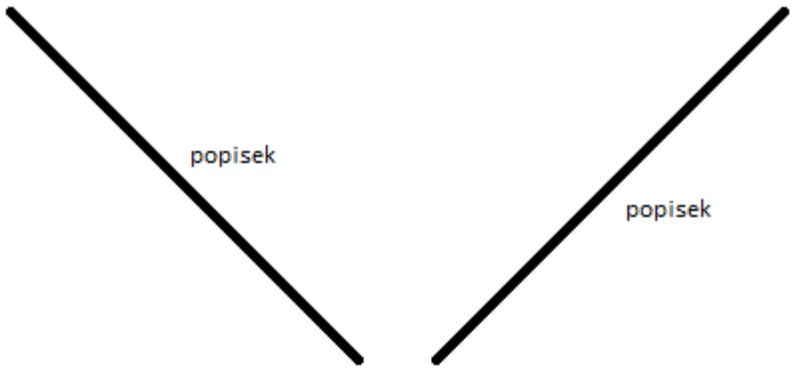
Z předchozích odstavců tedy víme, jak dostáváme do seznamu DFD prvků nové entity. Teď si již tedy můžeme popsat, jak budeme dané prvky a celý graf vykreslovat. Kreslení může nastat ve dvou stavech. Buď je aktivován některý z DFD tlačítek pro indikaci načrtávání daného prvku nebo není žádné tlačítko vybráno. Pokud je aktivováno načrtávání, potom nastavíme čárkovanou čáru a kreslíme načrtnutý DFD prvek. Po tomto náčrtku musíme samozřejmě ještě vykreslit ostatní prvky ze seznamu. Pokud není aktivováno načrtávání, pak jen vykreslíme prvky.

K vykreslování DFD prvků používám vlatní třídu `kresliObjekt()`. Ta na základě grafického kontextu, dvou souřadnicových bodů a typu prvku vykreslí do grafického kontextu předpřipravený objekt. K tomuto objektu je poté ještě nutno přidat popisek, o jaký typ prvku se jedná. O to se stará metoda `kresliDFDpopisek`. Ta má podobné parametry jako předchozí metoda, ale je ještě doplněna o popisek DFD prvku. Ten se vypisuje podle standardně předdefinovaných rozměrů grafických primitiv DFD prvků. Vyjímka je jen u datového toku. Tam nelze předem předvídat, jak bude tok vypadat. Opět mohou nastat čtyři situace, jak je tok definován (podobně jako u naslouchacích oblastí v kapitole 5.4.2). Tyto čtyři situace lze zredukovat na dvě situace a můžeme je vidět na obrázku 5.5.

Popisek je tak vždy vykreslován tak, aby nezasahoval do náčrtku toku.

Samotné vykreslování pak vypadá takto:

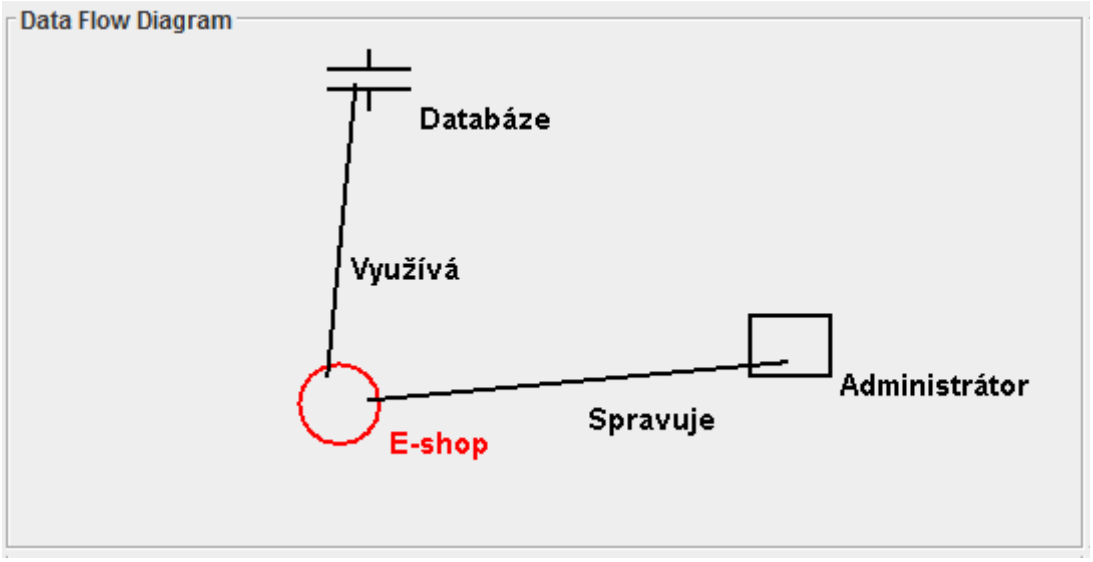
```
if(okno.jeNejakyDFDPrvekOznacen())
{
    gr.setStroke(carkovana);  // čárkovaná čára
    ...
    // načrtávám DFD objekt
    kresliObjekt(gr, new Point(x, y), new Point(konec), okno.typOznaceni);
}
gr.setStroke(plna);           // plná čára
if(!okno.model.seznamDFD.isEmpty()) // nejake objekty tam jsou
{
    // vykreslujeme
    for(int i = 0; i < okno.model.seznamDFD.size(); i++)
```



Obrázek 5.5: Správné vykreslení popisku bez zásahu do grafu

```
    {
        // kresli objekt a poté popisek
        kresliObjekt(gr, pom.bod1, pom.bod2, pom.typPrvku);
        kresliDFDpopisek(gr, pom.nazev, pom.typPrvku, pom.bod1, pom.bod2);
        ...
    }
}
```

Na následujícím obrázku 5.6 můžete vidět příklad velmi jednoduchého náčrtku DFD diagramu z prostředí aplikace. Na náčrtku není aktivní žádné tlačítko pro DFD prvky a aktuálně je myší označený prvek typu proces – E-shop.



Obrázek 5.6: Jednoduchý příklad DFD náčrtku s označenou entitou E-shop

5.5.2 Princip vykreslování chybového stromu

Vykreslovací plátno může pracovat ještě v druhém kontextu, konkrétně v kontextu analýzy pomocí chybových stromů. Vykreslovací plátno v tomto kontextu může obsahovat buď vykreslený chybový strom pro danou hrozbu nebo výpis hrozeb, které mohou pro daný DFD element nastat. Vše je ovlivněno tím, zda je na pracovní liště zakliknuto některé z tlačítek pro FT analýzu. Pokud není, tak jsou vypsány obecné informace. Pokud je, tak dochází k vykreslování chybového stromu. Obecné informace jsou zde vypisovány jako textové řetězce a na jejich vykreslování není nic složitého. Proto se rovnou zaměříme na princip, jak jsou vykreslovány chybové stromy.

Nejdříve si uveďme informace, ze kterých budeme vycházet a na kterých je vykreslování postaveno. Chybový strom je implementován pomocí třídy `FaultTree` a pro vykreslování je tedy důležité jeho správné vyextrahování z vnitřních datových struktur. K tomu slouží následující kód:

```
FTAAnalysis vykreslovanaAnalyza;
vykreslovanaAnalyza = okno.model.vratAnalyzuDleUid(uidZvyraznenyDFD);
kresliStrom(gr, vykreslovanaAnalyza.vratStrom(okno.typOznaceni));
```

Na základě vybraného DFD prvku (jeho identifikátoru) získáme konkrétní datovou strukturu FTA analýzy. Poté na základě typu označení (typ STRIDE hrozby) je vrácen konkrétní chybový strom zahrnující informace ke konkrétní analýze.

Počátek vykreslování stromu

Nejlepším způsobem, jak vykreslovat datovou strokturu stromu, je zvolit rekurzivní přístup ke kreslení. Jelikož jsem si pro uložení datové struktury zvolil raději seznam prvků než abstraktní datovou strukturu strom, je nutné definovat jakousi rozjezdovou metodu pro nastartování vykreslování a poté napojení rekurzivní metody.

Touto metodou je `kresliStrom()`. Invokací této metody se nejdřív provede pomocí metody `spocitejPravdepodobnosti()` spočítání pravděpodobností pro všechny uzly chybového stromu. Metoda vrátí speciální seznam. Více o těchto propočtech bude v kapitole 5.5.3. Dále se v této fázi provede propočet o velikosti pracovní plochy. Metoda totiž potřebuje znát hlavně šířku pracovní plochy. Na základě tohoto údaje bude při zanořování přidělována různá šířka pro další patra stromu. Kromě šířky jsou spočítány ještě dva body. Konkrétně se jedná o bod pro započetí vykreslování (umístěný v horní části pracovní plochy přesně uprostřed) a o bod pro připojení spodních pater. K tomuto bodu bude rekurzivní metoda popisovaná v následující části připojovat první poduzly kořenového prvku.

Vykreslování pomocí rekurzivní metody kresliPrvekStromu()

Hned na začátek popisu této metody si uveďme její hlavičku:

```
public void kresliPrvekStromu(Graphics2D gr,
                              FaultTree strom, PrvekFTA uzel,
                              Point kam, Point pripoj,
                              int sirka,
                              List<PravdPar> pravdepodobnosti)
```

Máme tedy k dispozici grafický kontext, do kterého se bude kreslit (`gr`), informace o stromové struktuře (`strom`), informace o aktuálně vykreslovaném uzlu (`uzel`), informace o bodech, kam se má uzel kreslit a kam má být graficky připojen (`kam` a `pripoj`), aktuální přidělenou šířku, kterou může vykreslovací algoritmus využít (`sirka`) a seznam spočítaných pravděpodobností (`pravdepodobnosti`).

Nejdříve tedy vykreslíme uzel do umístění `kam`. Na základě informací z objektu `uzel` získáme informaci o popisku a o logické operaci aplikované na potomky. Tato operace se graficky vykreslí pouze tehdy, pokud vykreslovaný uzel vůbec nějaké potomky má. Pokud nemá, místo operace se vykreslí symbol kolečka (viz návrh). Tímto jsme vykreslili aktuální uzel a je potřeba rekurzivně vykreslit potomky. Zkontrolujeme tedy, zda uzel má potomky. Pokud ano, tak si necháme vrátit seznam potomků. Spočítáme si je a hodnotu `sirka` si podělíme počtem těchto potomků a uložíme si je do speciální proměnné. Na základě hodnot `kam` a `pripoj` si spočítáme nové hodnoty, které v každém cyklu přepočítáváme. Cyklus má počet opakování podle počtu potomků a v každém cyklu se volá rekurzivně funkce pro vykreslování.

Kód této metody můžete vidět částečně v pseudotvaru zde (psoudopříkazy jsou v `<>`):

```
<vykresliUzel>;
<vykresliPopisek>;
<vykresliPravdepodobnost>;
if(strom.maPotomka(uzel))
{
    if(uzel.data.logika == Akce.FTA_AND)
        <vykresliLogickouOperaci(AND)>;
    else
        <vykresliLogickouOperaci(OR)>;
}
else
    <vykresliKolecko>;
if(strom.maPotomka(uzel))
{
    int pocetPotomku = strom.vratPotomky(uzel).size();
    int potrebnaSirka = sirka / pocetPotomku;
    Point kamNovy = <spocitejKamKreslit>;
    Point kamNovyPripoj = <spocitejKamPripojit>;
    for(int i = 0; i < strom.vratPotomky(uzel).size(); i++)
    {
        if(uzel.uid != -1)
            <vykresliPripoj>;
        kresliPrvekStromu(gr, strom,
                          strom.vratPotomky(uzel).get(i),
                          kamNovy, kamNovyPripoj,
                          potrebnaSirka, pravdepodobnosti);
        kamNovy.x += potrebnaSirka;
        kamNovyPripoj.x += potrebnaSirka;
    }
}
```

Pod pseudopříkazy `<vykresliPopisek>` a `<vykresliPravdepodobnost>` se skrývají skutečné metody `kresliPopisek()` a `kresliPravdepodobnost()`. Jejich volání je však ve zdrojovém kódu komplikovanější a proto jsem zde zvolil pseudotvar.

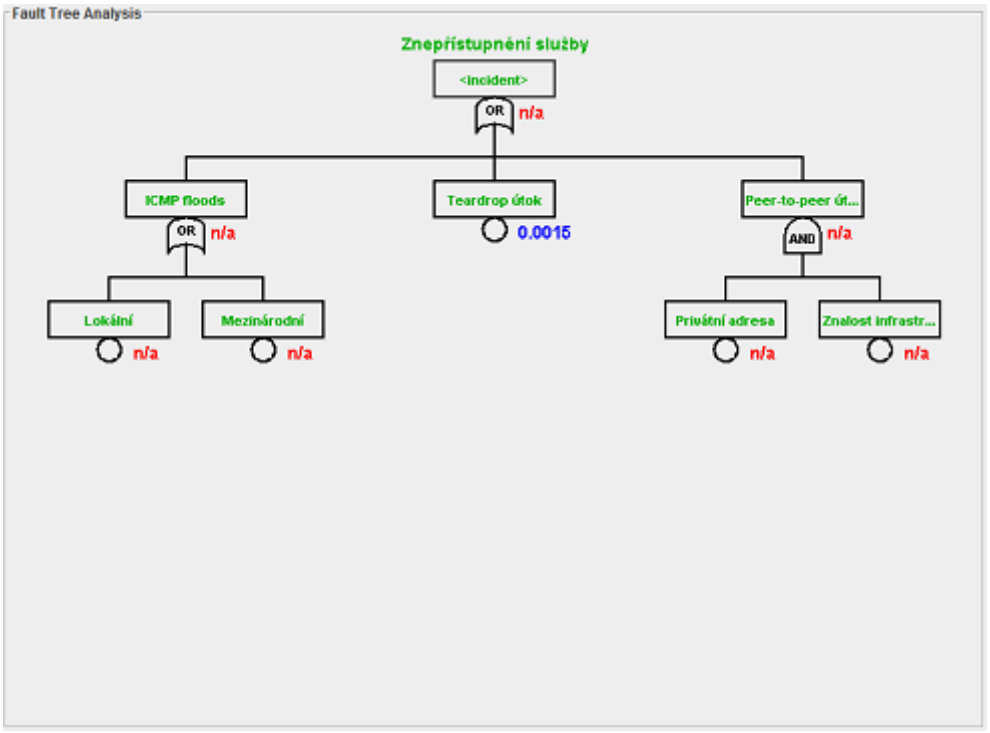
Metoda `kresliPopisek()` má tu vlastnost, že na základě předané hodnoty přes proměnnou typu `String` umí inteligentně vykreslit popisek uvnitř vykresleného uzlu. Umí si spočítat velikost prostoru pro popisek a pokud je popisek moc široký, tak postupně v cyklu od popisku odebírá z pravé strany jedno písmeno (a zároveň přidává napravo řetězec ...) a přepočítává šířku. Když nastane chvíle, že se již popisek do dané šířky vejde, je tento tvar vykreslen do dané oblasti.

Metoda `kresliPravdepodobnost()` na základě předané hodnoty typu `double` vyhodnotí, o jaký typ pravděpodobnosti se jedná. Zde mohou nastat 2 případy:

1. Hodnota je menší než 0.0 – potom je do daného umístění červenou barvou vykreslem řetězec n/a identifikující chybějící nebo nespočitatelnou hodnotu pravděpodobnosti.

2. Hodnota je větší nebo rovna 0.0 – potom je modře vykreslena hodnota pravděpodobnosti.

Na obrázku 5.7 identifikace hrozby Znepřístupnění služby pomocí chybového stromu. Z obrázku je patrné, že nejsou identifikovány všechny hrozby a nelze tedy spočítat celkovou hrozbu. Levý podstrom má kontext OR, protože je jedno jestli je hrozba lokální nebo mezinárodní, kdežto pravý podstrom má kontext AND, protože útočník musí útočit na privátní adresu a navíc musí znát infrastrukturu napadeného systému.



Obrázek 5.7: Identifikace hrozby Znepřístupnění služby pomocí FTA

5.5.3 Princip vyhodnocování chybového stromu

V této podkapitole se dostávám k samotnému jádru problému, tedy výpočtu hrozby z dílčích pravděpodobnostních údajů. V každém uzlu je počítána pravděpodobnost podle logické operace a podle zadané pravděpodobnosti. V části návrhu jsem tuto část přeskočil, protože jsem ještě přesně nevěděl, jaký algoritmus bude nakonec implementován. Nejdříve si zde tedy uvedeme trochu teorie.

Matematické podklady pro výpočet pravděpodobnostních údajů v uzlech

Základem celé stromové struktury, možnosti kompozice pravděpodobností a získání konečného výsledeku v kořenu stromu je nutnost znalosti dílčích pravděpodobností zanesených do všech listových uzlů chybového stromu. Tyto pravděpodobnosti není vůbec jednoduché zjistit a většinou se k tomuto problému přistupuje tak, že se tyto údaje buď odhadnou nebo se vychází ze statistických údajů, případně z incidentů, které již v minulosti nastaly.

Uveďme si nyní jednoduché příklady dvou stromů, kdy jeden implementuje stromovou operaci AND ($\wedge$) a druhý stromovou operaci OR ($\vee$). Záměrně zde zmiňuji slovo „stromovou“, protože stromová operace OR se od operace OR známé z logických obvodů či matematiky mírně odlišuje. Zohledňuje totiž i jistou možnou závislost prvků. Vzorce pro výpočet těchto operací vypadají následovně:

$$P(\wedge) = A \cdot B \cdot ... \cdot X \tag{5.1}$$

$$P(\vee) = A + B + ... + X - A \cdot B \cdot ... \cdot X \tag{5.2}$$

Pokud tedy chceme aplikovat stromovou operaci AND ($\wedge$), potom stačí dle vzorce 5.1 mezi sebou jednotlivé pravděpodobnosti od potomků vynásobit. Vzniká tím vztah, že všechny události musejí nastat současně, aby došlo ke kořenovému incidentu. Pokud chceme naopak aplikovat stromovou operaci OR ($\vee$), musíme dle vzorce 5.2 provést mezi všemi prvky součet a od tohoto součtu poté odečíst jejich součin. Zde tedy nastává situace, že musí být vždy zadány všechny pravděpodobnosti. U klasické operace OR by totiž stačilo mít zadanou pouze jednu hodnotu a výsledek by byl platný. Zde by se však ztrácel zisk daný součinem, který je nutno odečíst.

Metody využívané pro vyhodnocování stromu a jejich implementace

V této části se již rozepíši o implementaci vyhodnocování chybového stromu. Již jsem ukázal, jak aplikace vykresluje chybový strom a k tomu jsme použil i seznam spočítaných pravděpodobností. Nyní si ukážeme, jak jsem k těmto hodnotám přišel.

Ještě než přistoupím k popisu metody pro výpočet, je nutné zmínit existenci jednoduché třídy `PravdPar`. Tato třída je pouze datová třída a pomocí konstruktoru se vytvoří objekt obsahující dvě proměnné. Pracovně jsem to nazval pravděpodobnostní pár, kdy jeden objekt udržuje informaci o pravděpodobnosti pro uzel reprezentovaný jedinečným identifikátorem.

Stejně jako u vykreslovací metody je i počáteční metoda založená na rekurzivní funkci. Proto jsem si opět definoval startovací metodu, která nám vytvoří inicializační podmínky pro výpočet pravděpodobnosti ve všech uzlech. Základem je metoda `spocitejPravdepodobnosti()` s následující hlavičkou:

```
public List<PravdPar> spocitejPravdepodobnosti(FaultTree strom)
```

Výstupem této funkce tedy bude seznam pravděpodobností pro všechny uzly chybového stromu. Jak jste si již mohli všimnout, tento seznam byl již využit při vykreslování chybového stromu v předchozí kapitole. Tělo této metody spočívá v zavolání rekurzivní metody. Volání vypadá následovně:

```
return pravdepodobnostUzlu(strom.koren(), strom);
```

Rekurzivní metoda je založena na principu úplného zanoření až k listovým prvkům chybového stromu. Jdeme tedy postupně od kořene a u každého uzlu kontrolujeme přítomnost potomků. Pokud uzel má potomky, potom se dále zanořuje. Pokud metoda narazí na listový prvek, tak získá díky kontextu s předávanou proměnnou `strom` konkrétní údaj o pravděpodobnosti v listovém prvku. Na základě toho je vytvořen seznam o velikosti jednoho prvku. Tento prvek je objektem již zmiňované třídy `PravdPar` a vyjadřuje pravděpodobnost pro daný uzel. Tento seznam je vrácen zpět a začíná vynořování z rekurze. Vždy když jsou pro uzel dostupné všechny pravděpodobnosti, tak se provede výpočet a vynoření. Těmito dílčími výpočty se postupně rozrůstá vrácený seznam, až se dojde ke kořenovému prvku, kdy je seznam kompletní.

V následující ukázce kódu je vidět, jak celá rekurzivní metoda funguje. Opět jsem pro popis využil pseudokód a konkrétní výpočetní metody budou uvedeny záhy.

```
public List<PravdPar> pravdepodobnostUzlu(PrvekFTA uzel, FaultTree strom)
{
    List<PravdPar> vraceny = new ArrayList<>();
    List<PravdPar> dilci;
    if(strom.maPotomka(uzel))
    {
        List<PrvekFTA> potomci;
        potomci = strom.vratPotomky(uzel);
        for(int i = 0; i < potomci.size(); i++)
        {
            dilci = pravdepodobnostUzlu(potomci.get(i), strom);
            vraceny.addAll(dilci);
        }
        if(uzel.data.logika == Akce.FTA_AND)
            pravd = <spocitejPravdepodobnost(AND)>;
        else
            pravd = <spocitejPravdepodobnost(OR)>;
        PravdPar novy = new PravdPar(uzel.uid, pravd);
        // pripoj vysledek do seznamu
        vraceny.add(novy);
    }
    else
    {
        pravd = uzel.data.pravdepodobnost;
        if(pravd < 0.0 || pravd > 1.0)
            pravd = -1.0;
        PravdPar novy = new PravdPar(uzel.uid, pravd);
        // pripoj vysledek do seznamu
        vraceny.add(novy);
```

```
    }
    // vrat seznam
    return vraceny;
}
```

Záměrně jsem v tomto kódu použil pseudopříkazy pro výpočet pravděpodobnosti danou metodou, protože by to text znepřehlednilo. Každá z dvou následujících podkapitol se zaměří na jednu metodu.

Výpočet AND

Jak jsem již uvedl, výpočet AND je pouhý součin dílčích pravděpodobností. Zde je pouze nutno klást důraz na inicializaci. Při násobení se jako inicializační hodnota volí vždy hodnota 1.

```
...
pravd = 1;  // inicializace; 1 neovlivni vysledek
for(int i = 0; i < potomci.size(); i++)
{
    // ulozime si pravdepodobnost
    pom = pravdepDleSeznamu(potomci.get(i).uid, vraceny);
    // a zkontrolujeme na zapornost
    if(pom < 0.0)
    {
        pravd = -1.0;   // nastavime vysledek na zaporny a skoncime
        break;
    }
    pravd *= pom;
}
...
```

Výpočet OR

U výpočtu OR musíme kromě součtu dílčích pravděpodobností provést ještě odečtení jejich součinu. S touto operací může při nesprávné inicializaci vzniknout snadno fatální chyba. Kód vypadá následovně:

```
...
pravd = 0;  // inicializace; 0 neovlivni vysledek
double soucty = 0.0;
double nasobky = 1.0;
for(int i = 0; i < potomci.size(); i++)
{
    // ulozime si pravdepodobnost
    pom = pravdepDleSeznamu(potomci.get(i).uid, vraceny);
    if(pom < 0.0)       // a zkontrolujeme na zapornost
    {
        pravd = -2.0;   // nastavime vysledek na zaporny a skoncime
        break;
```

```
    }
    // vzorec je P(A) OR P(B) = P(A) + P(B) - P(A)*P(B)
    soucty += pom;
    nasobky *= pom;
    // jinak vypocitame vysledek
}
// zkontrolujeme, jestli nebylo nekde po ceste zaporne
if(pravd < -1.0)
{
    pravd = -1.0;
}
else
{
    if(potomci.size() == 1)
        pravd = soucty;
    else
        pravd = soucty - nasobky;
}
...
```

Chyba by v tomto případě mohla nastat, pokud by nebyl ošetřený stav, kdy bude počet potomků 1. Vezměme si například, že pravděpodobnost bude x. Provedeme tedy součty a výsledek bude x ($x + 0.0 = x$). Provedeme také násobky, kde výsledek bude také x ($x * 1.0 = x$). Pravděpodobnost bude $P(OR) = x - x = 0$. A tato chybová hodnota by se šířila celým stromem a konečná pravděpodobnost by byla 0.

5.5.4 Princip výběru FTA prvku pomocí roletového menu a akce

Jediným mechanismem pro práci a doplňování udajů do chybového stromu je výběr konkrétních uzlů pomocí roletového menu. Toto menu je umístěno na panelu vlastnosti a jeho implementace je rozdělena na několik částí. První část je samotná komponenta. Ta je umístěna ve třídě hlavní aplikace. Komponenta je inicializována pomocí jednorozměrného pole hodnot datového typu `String`. Toto pole je reprezentováno proměnnou `polozkyFTA`.

Jelikož lze pomocí tlačítek přidávat uzly do chybového stromu, potom je nutné zajistit aktuálnost položek tohoto roletového menu. K tomuto účelu je ve třídě `MainApp` implementována metoda `aktualizujCombo()`. Ta na základě invokace načte položky konrétního stromu z roletového menu. Před přidáním položek je ještě doplněna nultá položka označená řetězcem `---`, která signalizuje absenci vybraného uzlu.

Daný konkrétní objekt, který reprezentuje komponentu roletového menu má přiřazen posluchač na akce při výběru prvku z roletového menu. Pokud tato akce nastane, tak je volán konstruktor naší vnořené třídy, která implementuje akce spojené s výběrem prvku. Vnořená třída se nazývá `VyberPrvku` a je odvozená od standardní třídy `ItemListener`. Při volání konstruktoru přidáváme k volání typ roletového menu. Při implementaci jsem totiž počítal s tím, že budu implementovat vlastní roletové menu i pro výběr prvku u DFD náčrtku. Nakonec jsem od toho ale upustil, protože jsme chtěl náčrtek co nejvíce zjednodušit. Samotnou akci vykonávanou při výběru prvku obsluhuje přepsaná metoda `itemStateChanged()`. Pokud je tato metoda zavolána, potom víme že došlo k výběru prvku a získáme z roletového menu index vybrané položky. Tento index koresponduje s posunutým indexem v proměnné `polozkyFTA`. Posunutý index je kvůli nulté položce a od tohoto posunu je právě kvůli tomu

odečtena hodnota 1. Potom víme index prvku a na základě toho můžeme získat informace o prvku a na základě těchto informací vypsat údaje do připravených komponent a aktivovat související tlačítka, případně doplnit textová pole. Kód obsluhy vypadá ve zkratce takto:

```
...
// ziskam ID vybraneho prvku
index = comVyberPrvek.getSelectedIndex() - 1;
if(index > -1)  // osetreni inicialize, kdy padala aplikace
{
    FTAAnalysis analyza
          = model.vratAnalyzuDleUid(platno.uidZvyraznenyDFD);
    FaultTree strom = analyza.vratStrom(typOznaceni);
    aktualni = strom.seznam.get(index);
    idVybranyFTPrvekCombo = aktualni.uid;
    naplnInfoOFTA(aktualni);
    // pokud je vybran prvek korene, nelze tento prvek mazat
    if(idVybranyFTPrvekCombo == 0)
        tlSmazFTAPrvek.setEnabled(false);
    else
        tlSmazFTAPrvek.setEnabled(true);
    tlAktualizace.setEnabled(true);       // tlačítko
    tlPridatUzel.setEnabled(true);        // tlačítko
}
else
{
    // radsi to dame zpatky
    idVybranyFTPrvekCombo = -1;
    tlAktualizace.setEnabled(false);      // tlačítko
    tlSmazFTAPrvek.setEnabled(false);     // tlačítko
    tlPridatUzel.setEnabled(false);       // tlačítko

    fldNazevFTAHrozby.setText("---");     // textové pole
    fldPravdepVyskytu.setText("---");     // textové pole
}
...
```

5.6 Implementace ohodnocování celkového stavu modelu dle SDL

Jak jsem již uvedl v části návrhu, tak na panelu pro vlastnosti modelu analyzovaného pomocí naší aplikace se nacházejí dvě tlačítka pro kontrolu kvality modelu a pro celkový přehled hrozeb, které hrozí na daný systém. Následující podkapitoly osvětlí způsoby, jakými jsou dané informace získávány a jaké informace jsou prezentovány uživateli.

5.6.1 Analýza kvality modelu

V teoretické části věnované především procesu vývoje bezpečného softwaru SDL jsem uvedl kritéria, podle kterých lze určit kvalitu výsledného modelu. Pro zopakování si zde uvedeme

přehled jednotlivých úrovní.

0 – neexistuje Tímto se myslí, že ještě nebyla zvážena pro model žádná rizika. To znamená, že náčrtek neobsahuje žádné DFD prvky a pokud nějaké prvky obsahuje, tak potom nelze pomocí zavolání analýzy zjistit pravděpodobnost jakékoli hrozby u všech DFD prvků.

1 – nepřijatelný Pro takový model byla definována rizika. Pro každý DFD prvek je tedy provedena kontrola na identifikaci hrozeb a pokud u nějakého DFD prvku byl typ hrozby identifikován, potom model vyhovuje této úrovni.

2 – OK Kvalita takového modelu je definována tím, že pro každý DFD prvek lze identifikovat minimálně jednu hrozbu.

3 – dobrý Takový model musí splňovat podmínky úrovně 2 a navíc musí být splněno, že pro každý DFD prvek jsou definovány hrozby STIE (pokud mohou pro daný typ prvku nastat).

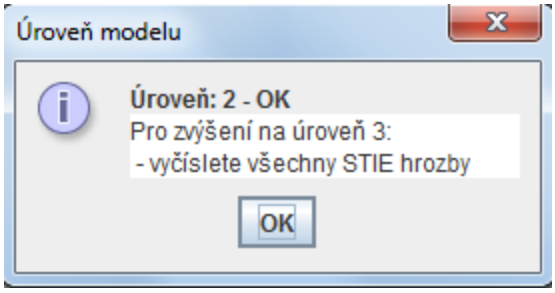
4 – výborný Takový model musí splňovat podmínky úrovně 2 a navíc musí být splněno, že pro každý DFD prvek jsou definovány hrozby STRIDE (pokud mohou pro daný typ prvku nastat).

K těmto úrovním jsem v aplikaci implementoval metody korespondující s podmínkami uvedenými u jednotlivých úrovní. Jedná se o metody `jeLevel1()`, `jeLevel2()`, `jeLevel3()` a `jeLevel4()`. Není zde potřeba implementovat metodu pro kontrolu úrovně 0, jelikož pokud není splněna podmínka pro úroveň 1, poté je model automaticky ohodnocen úrovní 0. Jelikož na sebe tyto úrovně navazují, je implementace celkového vyhodnocení provedena jako kaskáda podmínek. Metoda, která toto zahrnuje se jmenuje `zkontrolujModel()` a její pseudokód vypadá následovně:

```
...
if(!jeLevel1())
{
    <vypisUroven(0)>;
    <vypisNapoveduProUroven(1)>;
}   // je level 1, zkontrolujeme na level 2
else if(!jeLevel2())
{
    <vypisUroven(1)>;
    <vypisNapoveduProUroven(2)>;
}   // je level 2, zkontrolujeme na level 3
else if(!jeLevel3())
{
    <vypisUroven(2)>;
    <vypisNapoveduProUroven(3)>;
}   // je level 3, zkontrolujeme na level 4
else if(!jeLevel4())
{
    <vypisUroven(3)>;
    <vypisNapoveduProUroven(4)>;
```

```
}
else
    <vypisUroven(4)>;
...
```

Pseudopříkazy `vypisUroven` a `vypisNapoveduProUroven` jsou komponenty zobrazené na dialogu na konci této metody. Při invokaci této metody tedy bude celá vnitřní datová struktura vyhodnocena a například při splnění podmínek pro úroveň modelu 2 bude vypsána úroveň modelu jako dialog včetně nápovědy pro zvýšení na úroveň 3. Takový případ můžeme vidět na obrázku 5.8.



Obrázek 5.8: Informační dialog o úrovni modelu

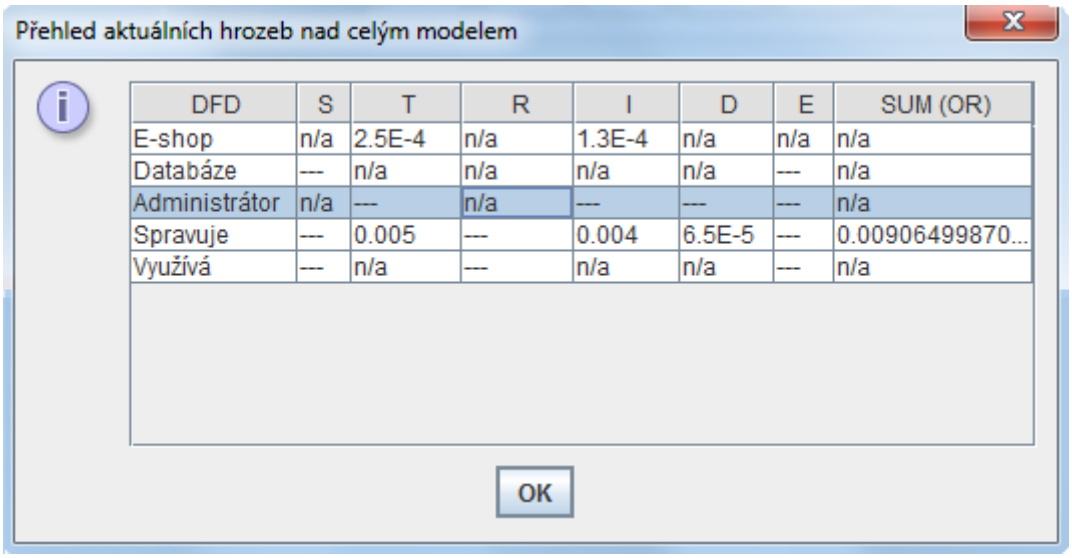
5.6.2 Přehled hrozeb

Jak jsem již uvedl v kapitole 4.3.5, která se zabývala návrhem, bylo potřeba nějak shrnout všechny údaje o pravděpodobnosti hrozeb, které mohou u systému nastat. K tomu jsem využil jednoduchou tabulku, do které jsem zanesl všechny pravděpodobnostní údaje (pokud byly spočitatelné nebo dostupné).

Implementace je zajištěna pomocí standardní funkce pro vytvoření dialogu. Dialog poté obsahuje tabulku a nyní si ukážeme, jak byla tabulka naplněna. V aplikaci zajišťuje zobrazení dialogu funkce `vypisPrehledHrozeb()`. Ta nejdříve načte pro všechny DFD prvky korespondující analýzy a na základě typu prvku DFD entity využívá funkce `vratkorenovouPravdepodobnost()` z třídy `Platno` pro spočítání celkové hodnoty pravděpodobnosti k dané hrozbě. Poté je ještě pro daný prvek zavolána metoda `pravdepodobnostDFD()` pro spočítání celkové hodnoty hrozby na daný DFD prvek. Tato hodnota je počítána jako stromová operace OR. Potom se již pouze vytvoří dialog a zobrazí se přehled daných hrozeb. Jelikož některé hrozby nemusejí být dostupné či nejsou ještě spočitatelné, tak je k výpisu využita metoda `vypisPravdepodobnost()`, která vstupní pravděpodobnostní hodnotu typu `double` převede na řetězec. Převod na řetězec pak funguje následovně:

- pokud je číslo větší než 100, potom se vypíše řetězec `---` identifikující neexistenci hrozby pro daný prvek (aplikace nastavuje tuto hodnotu na 101)
- pokud je čílo menší než 0.0, potom se jedná a definovanou, ale zatím nespočitatelnou hodnotu a systém vypíše řetězec `n/a` (aplikace nastavuje tuto hodnotu na -1.0)
- pokud nevyhovuje ani jedné z předchozích podmínek, potom se jedná o spočítanou hodnotu a vypíše se pravděpodobnostní údaj pomocí standardní metody `String.valueOf()`

Na obrázku 5.9 můžeme vidět vzhled dialogu pro zobrazení přehledu hrozeb. Je zde vidět, že některé hrozby jsou již definované všechny a lze pro ně spočítat celkovou hodnotu hrozby. Některé jsou prozatím nespočitatelné a jsou zde i prvky, u kterých některé hrozby nemohou nastat.



Obrázek 5.9: Informační dialog o přehledu hrozeb nad modelem

5.7 Implementace exportu a importu modelů vytvořených v aplikaci

Poslední důležitou částí aplikace je přidání možnosti importu a exportu modelů vytvořených v naší aplikaci. Pro import a export je nutné použít nějaký systém, který by správně a přehledně strukturoval data a umožnil jejich jednoduché ukládání do souboru a poté i načítání ze souboru.

Pro tyto potřeby jsem si zvolil open source knihovnu pro práci s xml dokumenty nazvanou *dom4j*[4]. Ke zdrojovým kódům je tato knihovna přiložena jako předkompilovaný balík `dom4j-1.6.1.jar`.

V následujících podkapitolách si popíšeme, co je všechno nutné udělat pro správný export a import datové stuktury pomocí xml dokumentu.

5.7.1 Vytvoření vnitřního objektu xml dokumentu

K zajištění všech akcí potřebných pro práci s xml dokumentem jsem implementoval vlastní třídu nazvanou `ImportExport`. Ta obsahuje několik důležitých proměnných sloužících pro uložení informací o xml dokumentu:

```
String nazevSouboru;
Model model;
Document document;
```

[4] http://dom4j.sourceforge.net/

```
Element koren;
```

Proměnná `nazevSouboru` slouží k uložení cesty k souboru včetně názvu. Máme tak hned přístup k informaci, kam bude soubor uložen nebo odkud má být soubor načten. Do proměnné `model` je načten odkaz na model, ze kterého budou brány informace pro uložení do xml dokumentu, případně kam budou zapisovány informace z xml dokumentu. Proměnná `document` slouží pro vnitřní reprezentaci xml struktury tak, jak ji definuje již dříve zmíněná knihovna dom4j. S tím také souvisí proměnná `koren`, do které je uložen odkaz na kořenový xml element (vnitřní reprezentace) z proměnné `document`.

Při vytvoření objektu třídy `ImportExport` je tedy zavolán konstruktor, který vytvoří základ pro xml objekt:

```
document    = DocumentHelper.createDocument();
koren       = document.addElement("model");
```

K další práci s xml objektem jsou definovány metody, které budou popsány v následujících podkapitolách podle toho, k jaké akci jsou použity.

5.7.2 Ukládání vnitřního modelu do xml dokumentu

K uložní do xml budeme nejdříve potřebovat mechanismy, jak z našeho objektu reprezentujícího model (`okno.model`) transformovat informace do objektu xml dokumentu (`document`). Pomocí metody z knihovny dmo4j potom bude tento objekt zapsán do souboru. K tomu máme v této třídě implementováno několik metod:

```
public void nactiModel(Model model)
private void vytvorFTACast(Element nadrazeny,
                           Vector<FTAAnalysis> seznamAnalyz)
private void pridejAnalyzu(Element nadrazeny, FTAAnalysis analyza)
public void ulozSoubor(String nazevSouboru)
```

Metoda `nactiModel()` slouží k tomu, aby vytvořila hlavní elementy xml dokumentu a poté pomocí metod `vytvorDFDCast()` a `vytvorFTACast()` do této vnitřní reprezentace zapsala hodnoty jednolivých prvků celého modelu. Po těchto akcích je potom již možné zavolat nad objektem metodu `ulozSoubor()`, která zajistí jeho správné uložení do souboru. Pseudokód načtení dokumentu do xml objektu vypadá takto:

```
...
this.model = model;
<vytvorZakladniXMLElement>
vytvorDFDCast(koren, model.seznamDFD);
vytvorFTACast(koren, model.seznamFTAAnalyz);
...
```

Metoda pro zápis do souboru pak vypadá takto:

```
OutputFormat format = OutputFormat.createPrettyPrint(); // formatovani
XMLWriter zapis = new XMLWriter(new FileWriter(nazevSouboru), format);
// zapisem dokument a zavreme zapisovac
zapis.write(document);
zapis.close();
```

Z hlavní aplikace tedy stačí provést následující akce a dojde k zapsání vnitřního modelu pomocí xml struktury do souboru. Předpokladem tohoto volání metod je již dříve získaná cesta k souboru:

```
ImportExport xml = new ImportExport();
xml.nactiModel(model);
xml.ulozSoubor(checkPripony(cesta));
```

5.7.3 Načítání z xml dokumetu do vnitřního modelu

Pokud chceme naopak načíst model z xml souboru, musíme nejdříve provést načtení xml souboru do vnitřní struktury reprezentující xml objekt a poté z tohoto objektu provést načtení dat do naší vnitřní struktury reprezentující model (`okno.model`).

Pro tyto akci máme v této třídě implementovány následující metody:

```
public boolean nactiSoubor(Model model, String cesta)
private void nactiDFD(Model model, Element dfd)
private void nactiFTA(Model model, Element fta)
```

Metoda `nactiSoubor()` nejdříve vymaže vnitřní reprezentaci modelu a poté načte pomocí čtečky knihovny dom4j xml dokument do xml objektu. Z tohoto objektu, konkrétně z kořenového elementu získá základní informace o modelu. Poté jsou zavolány metody `nactiDFD()` a `nactiFTA()`, které provedou načtení všech dat z xml objektu do vnitřní reprezentace modelu. Načtení těchto dat vypadá následovně:

```
model.vymazModel();
SAXReader reader = new SAXReader();
this.document = reader.read(new File(cesta));
<nacteniKorenovehoPrvku>;
nactiDFD(model, dfd);
nactiFTA(model, fta);
```

Metoda `nacteniFTA()` uvnitř své implementace ještě volá metodu `vratStromZXML()`. Jde o to, že struktura pro FTA část je mnohem složitější než struktura pro DFD část, a proto jsem pro přehlednost rozdělil načítání ještě do tohoto celku.

Pro načtení xml souboru do naší aplikace tedy stačí zavolat následující kód. Opět zde platí předpoklad již dříve získané cesty k souboru, tentokrát reprezentované proměnnou `nazev`.

```
ImportExport xml = new ImportExport();
resetPromennych();
xml.nactiSoubor(model, nazev);
```

Všimněme si ještě metody `resetPromennych()`. Jedná se o metodu implementovanou v třídě `MainApp` a slouží pro resetování všech stavových proměnných, pomocí kterých je řízen tok programu. Je totiž nutné se před načtením modelu vrátit do počátečního stavu aplikace.

5.7.4 Tvar xml dokumentu

Z předchozích podkapitol tedy již víme, jak programově vytvořit soubor reprezentující námi vytvořený model. Nyní se podíváme, jak vlastně takový soubor vypadá. Vycházel jsem hlavně z toho, aby struktura xml dokumentu přibližně odpovídala strukturování dat a objektů uvnitř datové struktury `Model`. Soubor s uloženým xml modelem může vypadat například takto:

```
<?xml version="1.0" encoding="UTF-8"?>

<model uidGenerator="3">
  <DFD>
    <prvek uid="0" typ="7" nazev="E-shop"
           p1x="178" p1y="130" p2x="0" p2y="0"/>
    <prvek uid="1" typ="10" nazev="Databáze"
           p1x="182" p1y="302" p2x="0" p2y="0"/>
    ...
  </DFD>
  <FTA>
    <analyza uid="0" typDFD="7">
      <s>
        <strom uidGenerator="3" typStride="201" uidDFDPrvku="0">
          <prvek uid="0" uidRodice="-1" popis="Krádež identity"
                 logika="401" pravdepodobnost="-1.0"/>
          <prvek uid="1" uidRodice="0" popis="Poslouchání sí'tové komunikace"
                 logika="400" pravdepodobnost="1.5E-4"/>
          ...
        </strom>
      </s>
      <t>
        ...
      </t>
      ...
    </analyza>
    <analyza uid="1" typDFD="10">
      ...
    </analyza>
  </FTA>
</model>
```

Tento xml soubor vznikl v korespondenci s obrázkem 5.6. Celý model je uzavřen v tagu `model`. Tento tag obsahuje atribut `uidGenerator`, protože musíme zajistit, aby se i po načtení modelu správně generovali nové unikátní identifikátory DFD prvků. Dále jsou v modelu vnořeny tagy `DFD` a `FTA`.

Tag DFD

Uvnitř tagu `DFD` máme elementy označující prvek. Tento prvek má přiřazeny atributy s informací o unikátním identifikátoru, typu DFD prvku, názvu DFD prvku, souřadnicích prvního a druhého bodu.

Tag FTA

Uvnitř tagu `FTA` máme elementy označující jednotlivé analýzy. Analýza má atribut označující unikátní identifikátor pro korespondenci s DFD prvkem a typ tohoto prvku. Uvnitř elementu pro analýzu jsou tagy reprezentující jednotlivé stromové struktury. U stromu je v atributech nutno uchovat informace o stavu generátoru pro generování unikátních identifikátorů pro nové prvky stromu, typ hrozby kterou daný strom sleduje a unikátní identifikátor DFD prvku. Jednotlivé prvku jsou pak zanořeny v elementu `strom` a mají v atributech informaci o identifikátoru, identifikátoru rodičovského uzlu, popis uzlu, logiku se kterou uzel pracuje a pravděpodobnost v daném uzlu.

Kapitola 6

Demonstrace použití aplikace a možnost dalšího vývoje

6.1 Hrozby dle STRIDE

V této kapitole jsou popsány základní hrozby a jak k nim může docházet. Účelem je, aby si návrhář mohl uvědomit, jak k danému incidentu (kořenové hrozbě) může docházet a jaké dílčí hrozby a jaké souvislosti vedou k vyvolání daného incidentu. Informace o útocích byly částečně čerpány z [1].

6.1.1 Spoofing Identity – krádež identity

Identita je něco, co nás odlišuje od ostatních lidí. V počítačovém světě se jedná například o přezdívku na fóru, zákaznické číslo v nějakém elektronickém obchodě, bankovní údaje a další. **Krádež identity** je definována jako nedovolené shromažďování a používání osobních údajů, které jsou obvykle využívány za účelem kriminální činnosti. Zjednodušeně lze tedy říci, že se jedná o odcizení elektronických údajů.

Dílčí činnosti vedoucí ke krádeži identity jsou:

- phishing – lstivé vylákání osobních údajů
- pharming – napadení DNS a přepsání IP adresy k přesměrování oběti na podvodnou stránku
- skimming – neoprávněné kopírování dat
- hacking – neoprávněné vniknutí do počítače
- a další…

6.1.2 Tampering – narušení aplikace modifikací dat nebo kódu

Anglický termín **tampering** označuje útok, kdy je nějaký produkt upravený útočníkem tak, aby spotřebiteli způsobil nějakou škodu. Z hlediska informačních technologií se jedná o modifikaci zdrojového kódu nebo chování aplikace tak, aby vykonávala nějaké nechtěné činnosti.

Dílčí činnosti vedoucí k této hrozbě:

- vir – zdrojový kód je modifikován nebo doplněn zdrojovým kódem viru (prodlužující/přepisující/duplicitní viry)
- přepis klient-server požadavku – napadení při komunikaci
- závadný soubor – u aplikací s možností importu dat využití neošetřeného vstupu závadného souboru (pád aplikace)

Nejrozšířenější jsou v této oblasti viry. Dělíme je na prodlužující (škodlivý kód přidávají na konec původního zrojového kódu), přepisující (přepíší hlavičku zdrojového kódu a zamezí tak spuštění původního kódu a dojde ke spuštění škodlivého kódu) a duplicitní (napadají exe soubory tak, že vytvoří v adresáři spustitelný soubor se škodlivým kódem se stejným jménem a s příponou com, která má při spouštění přednost).

6.1.3 Repudiation – zamezení logování a sledování uživatelských akcí

Jedná se o útok, kdy aplikace nemůže správně zalogovávat informace o svém běhu nebo jsou narušeny uživatelské akce. Zalogovaná data jsou poté označena jako nepravdivá a nedůvěryhodná.

Dílčí činnosti vedoucí k této hrozbě:

- neoprávněný přístup – aplikaci je podstrčena disková cesta, kam daná aplikace nemá povolen zápis a není prováděno logování
- napadení funkcí – modifikace kódu funkcí, které ukládají data do logů tak, že ukládají nepravdivé údaje

6.1.4 Information Disclosure – zpřístupnění informací neoprávněným osobám

Tento útok dovoluje útočníkovi získat hodnotné informace o systému. Je tedy nutné tyto informace chránit, protože mohou útočníkovi pomoci při napadení systému.

Dílčí hrozby:

- odchycení HTTP hlavičky – útočník může z http hlavičky získat informace o prohlížeči a další informace
- získaní informací o politikách – útočník může získat informace o uživatelských politikách a najít v nich slabinu pro útok
- získání memory dumpu[1] – z memory dumpu je možné získat informace o konkrétním systému
- získání koncové adresy – útočník může tuto informaci využít k napadení síťovou cestou
- odchyt certifikátu – pokud je prováděn přenos certifikátu nešifrovanou cestou, pak ho lze odchytit a poté využít při sledování šifrované komunikace
- získání informací o závislosti služeb – pokud je služba závislá na jiné službě, tak lze ovlivnit chod této služby napadením té služby, na které je závislá

[1] memory dump – při pádu aplikace je vytvořen soubor o posledním stavu aplikace před pádem

6.1.5 Denial of service – znepřístupnění služby

Jedná se o útok, kdy je pomocí zahlcení služby znepřístupněna aplikace nebo je její správný chod vážně narušen či zpomalen. Cílem tohoto útoku je buď dosáhnout neustálého restartování dané služby nebo zpomalení dané služby tak, že je nedostupná či je odezva od služby velice dlouhá.

Dílčí typy útoků:

- ICMP floods – na základě špatného nastavení všechny počítače v síti posílají broadcast na všechny počítače a dojde tak k zahlcení jejich komunikace (smurf attack) nebo ping-flood či SYN-flood
- Teardrop útok – zaslání IP fragmentu s překrývajícím se příliš velkým nákladem dat na cílový počítač. Na cílovém počítači se poté při rekonstrukci dat může dostat OS do kritického bodu a může dojít až k pádu OS.
- Peer-to-peer útok – jde o využití chyby v P2P klientech, kdy je možné připojováním a odpojováním zahltit uživatele požadavky.
- Nukes – jedná se o speciální paket, který při zpracování vyústí v pád OS.

6.1.6 Elevation of Privilege – získání administrátorských práv

Tento typ útoků má za následek to, že útočník získá vyšší oprávnění než které mu byly původně přiděleny. Příkladem je získání práv ke čtení a zápisu, i když měl útočník v počáteční fázi pouze ke čtení.

Typy útoků:

- Přepnutí identity bez bezpečnostního kontextu – speciální případ chyby u použití .NET Framework 3.0; identita se změní na uživatelskou, ale práva zůstanou administrátorská.
- Reset expirace politiky na vysokou hodnotu – při resetování politiky může dojít k překročení (či překrytí) úrovně práv a útočník tak může získat další práva.
- Nesynchronizované certifikáty – klient používá jiný certifikát než server.

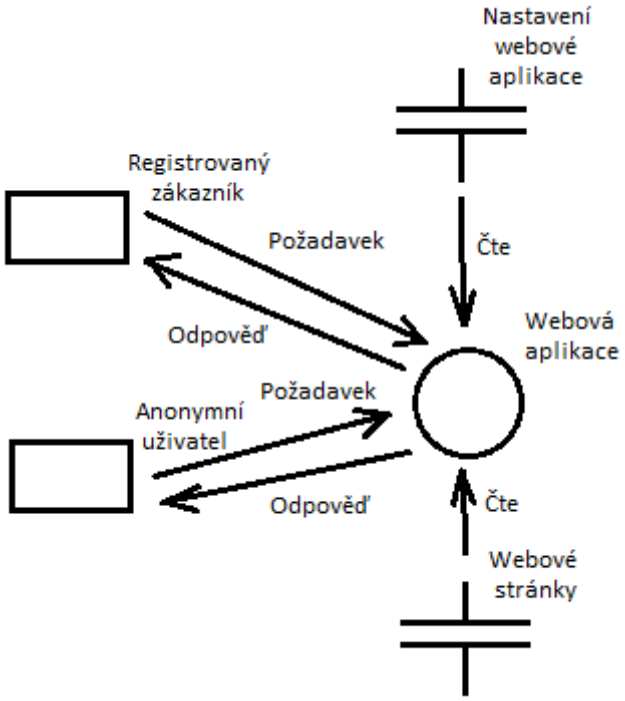
6.2 Typový příklad

V této kapitole uvádím typový příklad pro využití naší aplikace k analyzování hrozeb hrozících na daný systém.

Aby bylo možné udělat náčrtek systému, potřebujeme nějaký jednoduchý systém na demonstraci. V teoretické části jsem uvedl jednoduchý systém, kde byl načrtnut elektronický obchod. Pro naše účely si tento systém zjednodušíme a bude vypadat přibližně jako na obrázku 6.1.

Jak je vidět na obrázku, tak náčrtek obsahuje 2 entity typu uživatel, 2 entity typu datové úložiště a 1 entitu typu proces. Pokud si tento náčrtek překreslíme do naší aplikace, tak výsledek bude vypadat přibližně jako na obrázku 6.2.

Jak vidíme, tak v aplikaci můžeme slučovat toky, pokud směřují ke stejným entitám. Takto jsme redukovali také počet nutných DFD prvků k analýze. Respektive pokud budeme



Obrázek 6.1: Náčrtek pro elektronický obchod [5]

analyzovat datový tok *Komunikace 1*, potom bude hrozit stejné riziko pro toky *požadavek* a *odpověď* z původního náčrtku.

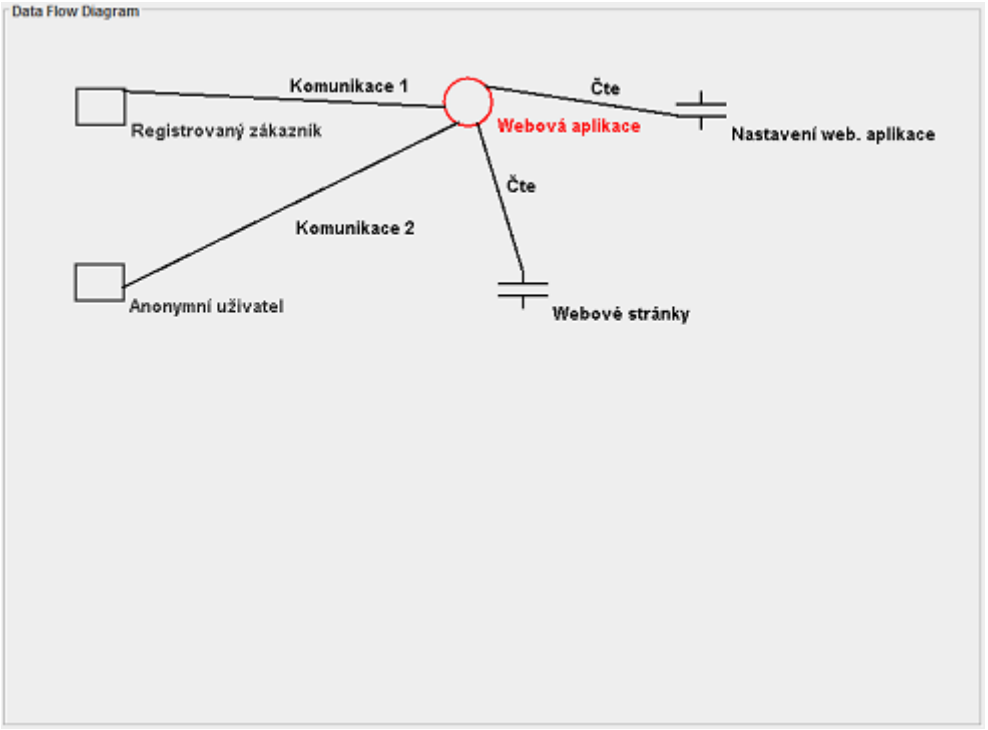
Nyní tedy máme nakreslený náčrtek. Nyní si můžeme kliknout na tlačítko *Úroveň modelu* pro ověření úrovně. Vyskočí dialog vykreslený na obrázku 6.3.

To nám ukázalo, že pouze DFD náčrtek nestačí a zároveň máme v dialogu nápovědu, že pokud chceme zvýšit kvalitu modelu, musíme definovat alespoň nějaká rizika.

Označíme si proto entitu *webová aplikace* a tím se nám zpřístupní tlačítko pro přepnutí do pohledu pro FTA analýzu. Na pracovní plochu se nám vypíší informace o hrozících incidentech na daný DFD prvek. V tomto případě se jedná o prvek typu proces a pro ten hrozí všechny typy STRIDE hrozeb. Zároveň na této obrazovce vidíme informace o tom, že všechny stromové struktury zatím nehlásí žádná identifikovaná rizika a u všech jsou výsledné pravděpodobnosti nedefinované (n/a). Klikneme tedy na tlačítko s písmenem S. Díky tomu se nám zobrazí pohled s kořenovým prvkem reprezentujícím kořenový incident, v tomto případě incident *Krádež identity*. Zároveň se nám aktivoval panel pro práci se stromem. Nyní tedy můžeme vytvářet uzly stromu.

Vyjdeme-li z návrhu v kapitole popisující STRIDE hrozby, můžeme se zamyslet, jak by pro proces *webová aplikace* mohlo dojít ke krádeži identity. Pro proces určitě hrozí hacking, protože zdatný útočník by mohl neoprávněně vniknout na cizí systém. Jelikož jde o webovou aplikaci, určitě hrozí i útoky typu phising a pharming. Přidáme tedy tyto 3 typy hrozby do stromu. Vždy vybereme pomocí roletového menu kořenový uzel, klikneme na tlačítko pro přidání. Kořenový uzel má automaticky nastavený kontext OR.

Nyní máme definovaný chybový strom a třech poduzlech. Pokud nyní klikneme na ověření modelu, tak vidíme, že se úroveň modelu zlepšila na úroveň 1. Zároveň se nám zobrazila nápověda, jak dosáhnout úrovně 2 – musíme definovat pro každý prvek alespoň



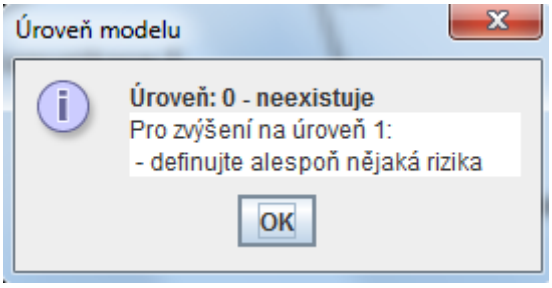
Obrázek 6.2: Překreslený náčrtek v aplikaci

jednu hrozbu. Všimněme si také, že není třeba ještě zadávat žádné pravděpodobnostní údaje a kvalita modelu si již zvedá. Je to dáno tím, že požadavek na úroveň jedna je, aby si byl návrhář vědom alespoň nějakých hrozeb na systém.

Stejným způsobem jako pro entitu *Webová aplikace* tedy definujeme pro každý DFD prvek nějakou hrozbu. Klikněme tedy na tlačítko pro přepnutí mezi DFD a FTA analýzou, vybereme prvek a přidáme nějaké hrozby. Až tak učiníme, tak klikneme opět na tlačítko pro ověření úrovně modelu. Nyní se nám zobrazí, že opět došlo ke zvýšení modelu, a to na úroveň 2. Nápověda pro úroveň 3 je, že je nutné identifikovat všechny hrozby STIE. Je tedy nutné projít model a definovat všechny tyto hrozby. Tak můžeme postupovat až k maximální kvalitě modelu – úrovni 4.

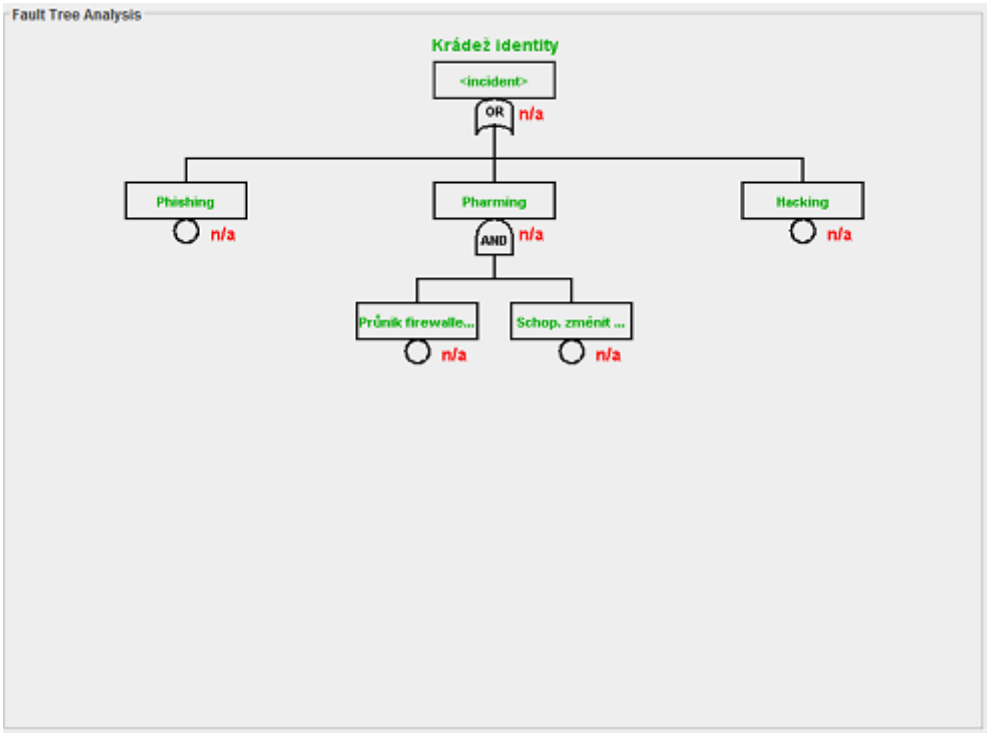
K vyčíslování jednotlivých incidentů si toho řekneme více zde. Pravděpodobnostní údaje bohužel není tak lehké určit a mnohdy je k tomu potřeba vynaložit velké finanční prostředky a čas, aby bylo možné získat data žádané přesnosti. Získávání těchto dat není předmětem této práce a proto jsme je náhodně zvolili. Je jasné, že musí jít o desetinné číslo mezi hodnotou 0 a 1.

Vraťme se nyní k analýze hrozby S u DFD prvku *Webová aplikace*. Definovali jsme si 3 hrozby a nyní můžeme buď přímo zadat pro dílčí hrozby pravděpodobnost jejich výskytu nebo můžeme definovat další dílčí hrozby, vedoucí k této hrozbě. Takto můžeme definovat kaskádu hrozeb. Definujme si tedy alespoň nějaké podhrozby, například u útoku pharming. K tomuto útoku je nutno pomocí DNS podstrčit doménové jméno. Aby toho mohl útočník dosáhnout, řekněme že musí mít schopnost nastavit na cílovém počítači adresu DNS serveru a musí projít firewallem. Tyto 2 hrozby tedy definujeme jako podhrozby uzlu FTA stromu, který je označen popiskem *Pharming*.



Obrázek 6.3: Ověření úrovně modelu

Výsledný strom tedy může vypadat jako na obrázku 6.4.



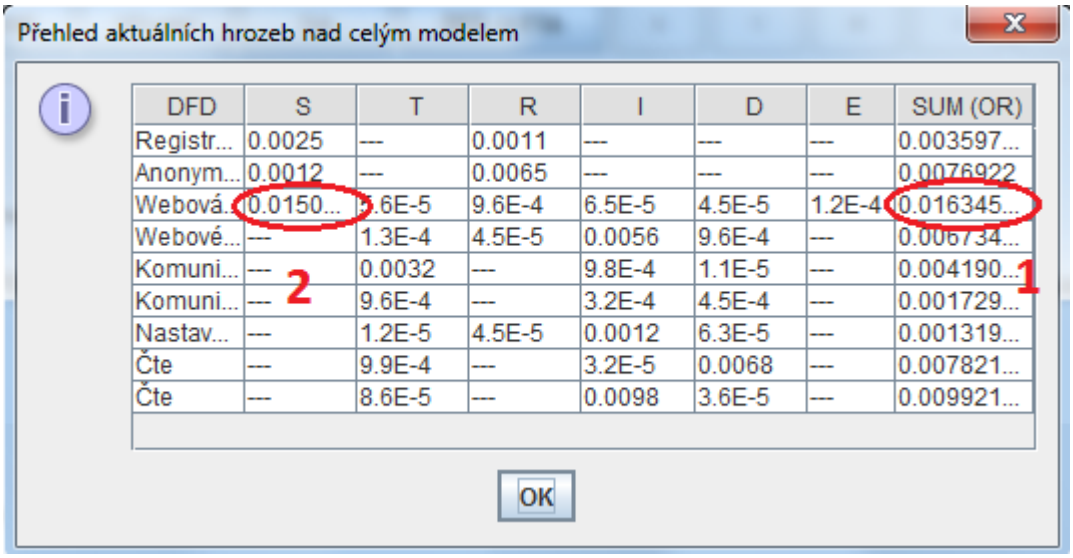
Obrázek 6.4: Chybový strom pro incident Krádež identity

Princip této analýzy je v tom, že pokud neznáme pravděpodobnost hrozby na systém, potom můžeme tu hrozbu dělit na dílčí hrozby v různém kontextu (AND/OR) a rozkouskovat tak problém na menší celky, pro něž je získání pravděpodobnostního údaje jednoduší.

Ve výsledku je tedy cílem získání přehledu o hrozbách na daný DFD prvek, vybrat hrozbu s nejvyšší pravděpodobností a pro tu si zobrazit chybový strom. Na jeho základě si poté identifikovat hrozby, které nejvíce ovlivňují incident a na možnost této hrozby si dát při implementaci systému největší pozor. Výstupem však nemusí být pouze to, na jaké zranitelnosti klást větší důraz při implementaci, ale také způsob, jakým zabezpečit systém, na kterém naše aplikace poběží.

Přejděme teď ale dále a vyjděme z případu, kdy máme celý model doplněný. Potom je dobré kliknout na tlačítko pro přehled hrzob. Zobrazí se nám dialog s tabulkou, kde jsou

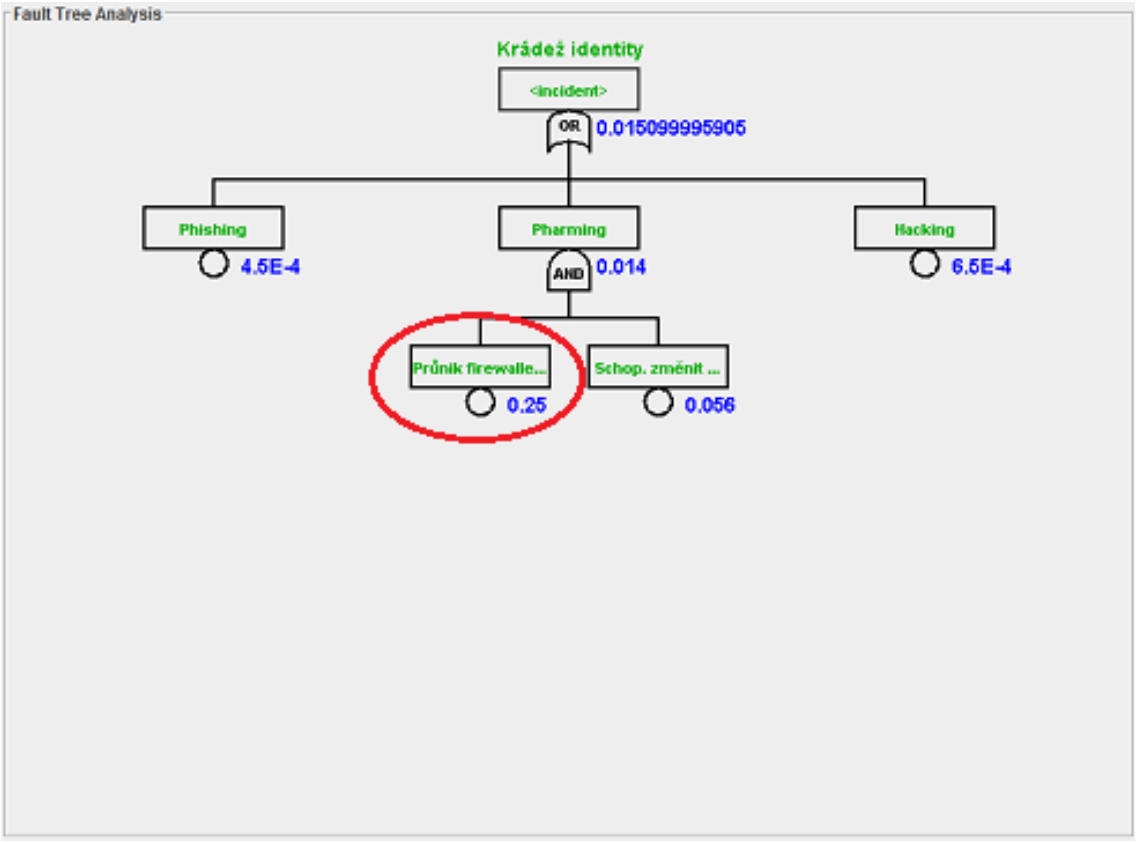
zaneseny všechny hrozby a všechny vypočítané chybové stromy. Tento dialog můžete vidět na obrázku 6.5.



Obrázek 6.5: Přehled hrozeb a jeho analýza

V přehledu hrozeb vidíme všechny dílčí výpočty. Zaměříme se na DFD prvek, který má největší pravděpodobnost výskytu incidentu. Suma hrozeb vyjadřuje celkovou napadnutelnost DFD prvku. Největší napadnutelnost má dle tabulky (na obrázku označeno číslem 1) prvek *Webová aplikace*. Ve stejném řádku se zaměříme na sloupec, který má největší podíl na zisku tak vysoké napadnutelnosti. Jedná se o incident *Krádež identity*. Zavřeme tedy dialogové okno a přejdeme do DFD náčrtku. Označíme si entitu *Webová aplikace* a přepneme na FTA analýzu. Dále klikneme na tlačítko S pro analýzu incidentu *Krádež identity*. Zobrazí se nám chybový strom. Vizuální analýzou zjistíme, který dílčí incident přispívá nejvíce k zisku tak vysoké napadnutelnosti systému. Takový dílčí incident je označen na obrázku 6.6.

Takto lze postupně analyzovat všechny velké hrozby na systém a navrhovat ať už úpravy konkrétního systému nebo později i úpravy prostředí, ve kterém bude zkoumaný systém nasazen.



Obrázek 6.6: Chybový strom včetně výpočtů s označenou kritickou hrozbou

6.3 Použití v reálném prostředí

Aplikace je založena na procesu tvorby bezpečného softwaru – SDL. Proces má několik fází, přičemž všechny fáze jsou řešeny na úrovni dokumentace, informačních schůzek ve firmách a využívání systému pro sledování chyb. Pro analyzování rizik a jejich ohodnocování v tomto směru zatím neexistuje žádný softwarový nástroj. Pokusil jsem se vytvořit nástroj, který by nejen rizika identifikoval (SDL rizika pouze identifikuje a nevyjadřuje je matematickou hodnotou), ale který by je také uměl vyjádřit matematicky a dílčí hrozby dával do kontextu a vztahů. Vyčíslování jsem tedy založil na principech analýzy pomocí chybových stromů. Pro uživatele, kteří mají data o četnosti výskytu různých hrozeb na systém a jsou seznámeni s analýzou pomocí těchto stromů, je tento nástroj ideálním řešením.

Abych to tedy shrnul, tak pokud je uživatel seznámem s principy FT analýzy a má přístup k matematickým (pravděpodobnostním) datům o výskytech hrozeb v systémech, potom může nástroj efektivně a rychle identifikovat hrozby v širším kontextu a více do-hloubky.

6.4 Možnost dalšího vývoje

Jak to již ve světe softwaru chodí, každá aplikace se dá vždy vylepšovat a stále je možné přidávat další rozšiřující funkce. Nejinak tomu je i u naší aplikace.

Analýza rizik je velice komplexní téma a existuje spousta metod, jak k řešení analýzy přistupovat. Já jsem si zvolil přístup, kdy jsem se zaměřil hlavně na napadnutelnost daných částí systému, případně napadnutelnost prostředí, ve kterém bude systém nasazen. Zaměřil jsme se tedy hlavně na matematický způsob analýzy rizik a především na vyčíslování pravděpodobnosti, se kterou mohou dané hrozby nastat.

Jako rozšíření k danému způsobu by bylo možné zvážit implementaci zvažování dopadů na systém z hlediska ekonomické stránky. Princip by spočíval v detekci nejen pravděpodobnosti výskytu incidentu na systém, ale také finanční ztráty, kterou by incident způsobil. Tím by vznikl úplně jiný pohled na problém, kdy by nemusel být analyzován nekritičtější prvek navrženého systému, ale prvek, který by v případě výskytu incidentu způsobil největší finanční ztráty.

U zadávání dílčích hrozeb se také vyskytla možnost, kdy některé hrozby mohou nastat v samotné aplikaci a jiné až při nasazení hotového produktu. Jako rozšíření by tedy také šlo implementovat příznaky u hrozeb, které mají vliv na vývoj samotného produktu. Model by byl tedy dvoustavový – buď ve stavu návrhu nebo ve stavu nasazení. Ve stavu návrhu by se braly v potaz při analýze pouze ty hrozby, které je možné zmírnit, či které je možné úplně odstranit. Tím by vznikala zpětná vazba pro vývojový tým a mohlo by se zamezit hrozbám na systém ještě před vydáním produktu. Ve stavu nasazení by se pak braly v potaz všechny hrozby.

Kapitola 7

Závěr

Cílem tohoto textu bylo vytvořit softwarový nástroj pro možnosti analýzy rizik v informační bezpečnosti. Nejdříve byl představen proces životního cyklu vývoje bezpečného softwaru, následovaný podrobným rozborem analýzy rizik tak, jak k ní přistupuje výše zmíněný proces. Tato analýza je ale založena spíše na ukazatelích a kvalitativních metrikách, a proto jsem se ji rozhodl rozšířit o další analýzu. Jedná se o analýzu pomocí chybových stromů. Tuto část se povedlo vhodně zakomponovat do analýzy rizik dle SDL a vznikl návrh aplikace.

Návrh byl rozdělen na část zabývající se grafickým uživatelským rozhraním, poté na návrh efektivních vnitřních struktur pro uchovávání informací o modelu a nakonec ještě návrh řízení toku programu.

Bezprostředně za návrhem byla popsána kompletní implementace aplikace. Implementace byla rozdělena na vývoj grafického uživatelského rozhraní, vnitřních datových struktur a velká část byla věnována implementaci pracovní plochy. Ta je v náčrtkové fázi plně interaktivní a ovladatelná pomocí myši a v analytické části se ovládá pomocí bočního panelu. V korespondenci s návrhem podle SDL je v této kapitole také popsána implementace vyhodocování kvality modelu, založené na kvalitativních metrikách, popsaných v úvodních teoretických kapitolách. Z matematického hlediska je doplněna implementace přehledu hrozeb napomáhající k rychlé a efektivní identifikaci největších potencionálních hrozeb. Kapitola je uzavřena popisem implementace modulu pro importování a exportování modelů pomocí XML struktury.

Závěrečná kapitola shrnuje všechny údaje a demonstruje použití aplikace na typovém příkladu. Jsou zde uvedeny přesné kroky, jak postupovat při tvorbě modelu a zlepšování jeho úrovně. Z popisu dále vyplývá, jakým způsobem je možné identifikovat nejzávažnější hrozby. Závěr této kapitoly obsahuje informaci o možném využití tohoto nástroje v praxi a o jeho dalších možných implementačních rozšířeních.

Literatura

[1] *MSDN Library* [online]. 1995 [cit. 2012-05-16], dostupné na URL http://msdn.microsoft.com/en-us/library/.

[2] EDWARDS, W.; MILES, R. F.; WINTERFELDT, D. V.: *Advances in Decision Analysis: From Foundation to Applications*. Cambridge University Press, 2007, iSBN 978-0-521-68230-5.

[3] Ericson, C. A.: *Fault Tree Analysis* [online]. 1999 [cit. 2012-01-05], dostupné na URL http://www.fault-tree.net/papers/ericson-fta-tutorial.pdf.

[4] GARLICK, A.: *Estimating Risk: A Management Approach*. Gover Publishing Limited, 2007, iSBN 978-0-566-08776-9.

[5] HOWARD, M.; LIPNER, S.: *The Security Development Lifecycle*. Microsoft Press, 2006, iSBN 978-0-7356-2214-2.

[6] Kreslíková, J.: *Řízení rizik v projektu* [online]. 2012 [cit. 2012-05-08], dostupné na URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/MPR-IT/lectures/rok-2011/10s_rizeni%20rizik.pdf.

[7] LIMNIOS, N.: *Fault Trees*. John Wiley & Sons, 2010, iSBN 978-0-470-39461-8.

[8] ROBERTS, N. H.: *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, 1981, iSBN 0-16-005582-2.

Příloha A

Obsah CD

CD nosič obsahuje následující data:

- app – aplikace
  - src – zdrojové soubory s instrukcemi pro překlad v souboru readme.txt
  - bin – spustitelná aplikace
  -
- doc – dokumentace
  - scr – veškeré zdrojové LaTeXsoubory a obrázky použité v dokumentaci
  - xplise00.pdf – text diplomové práce
- man – manuál
- soft – software potřebný pro spuštění aplikace